DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

The Effectiveness of GitHub's Security Interventions on Code Security

Jonas Höbenreich

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

The Effectiveness of GitHub's Security Interventions on Code Security

Die Wirksamkeit der Sicherheitsmaßnahmen von GitHub auf die Codesicherheit

Author:	Jonas Höbenreich
Supervisor:	Prof. Jens Großklags, Ph.D.
Advisor:	Felix Fischer
Submission Date:	June 15, 2022

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, June 15, 2022 Jonas Höbenreich

Acknowledgments

The authors gratefully acknowledge the computational and data resources provided by the Leibniz Supercomputing Centre (www.lrz.de).

1 Abstract

After GitHub announced its acquisition of Semmle in 2019, a code scanning workflow based on Semmle's semantic code analysis engine CodeQL was introduced in 2020, enabling developers of open source projects to automatically scan their GitHub repository for security vulnerabilities.

For this thesis, the JavaScript code in 206,491 snapshots from 8,931 open source repositories was successfully analyzed for improper neutralization of input during web page generation (CWE-79 vulnerabilities). These repositories are hosted on GitHub and have security scanning enabled. **After enabling CodeQL, the number of issues decreased by about 2%; in absolute terms, an average of approximately 0.04 fixes were applied per repository.** No vulnerability was identified in 7,337 repositories (82%) in the period of 45 days before and 45 days after CodeQL activation. However, 1,594 projects (18%) contained at least one problem. The effect of CodeQL on these projects differs greatly. In 6% of these projects, the number of problems was reduced by at least 95% after CodeQL was activated, and another 10% of repositories also improved. No change was observed in 68% of the affected projects, while 16% of the projects showed worse performance after CodeQL was activated. Overall, the analyses show that CodeQL improved the average security of the projects, but the greatest reduction in vulnerabilities occurred shortly after CodeQL scanning was enabled.

Contents

2 Introduction

2.1 Software Security in Open Source Projects

To operate software systems securely, companies have a large number of options at their disposal. These include peer review among colleagues (manual code inspection), static and dynamic code analyzers, penetration testing, intrusion detection systems, bug bounty programs and red, blue and purple teams. In addition, large companies and governmental organizations also commission external contractors to check their infrastructure for possible weak points (Alomar et al. 2020). Furthermore, there are numerous best practices that companies are strongly encouraged to follow. These include securing the network and all privileged or administrative access points with 2 factor authentication, regular security updates and security training for staff (CISA 2022).

Nevertheless, malicious actors have managed to bypass security systems in multiple high-profile cases in the recent past, resulting in service disruption, data breaches, fraud and theft of intellectual property, among other things. This shows that the efforts made so far are obviously not enough. It is worrying that critical infrastructure such as the energy sector is often affected (CSIS 2022; NCSC 2018).

The vulnerability CVE-2021-44228 in the popular open source library Apache Log4j 2 is particularly noteworthy.[1] Although at the time the vulnerability became known, the repository had only about 1,300 stars on GitHub[2]; it is a widely used component. This critical vulnerability led to numerous attacks and affected organizations of all sizes. Among others, at least six U.S. states and the Belgium Ministry of Defense were affected. The latter was forced to shut down parts of their computer network for several days, including the mail system (CSIS 2022).

At the 2022 Open Source Software Security Summit II in Washington, DC, a 10-point action plan was developed to increase the security and reliance of open source software (OpenSSF 2022). This plan acknowledges that maintainers of open source software often have significantly fewer resources available than corporations and government organizations. Nevertheless, security vulnerabilities in open source software impose a serious threat to organizations and individuals, as the Log4j example illustrates. A

[1] https://www.cve.org/CVERecord?id=CVE-2021-44228
[2] https://github.com/apache/logging-log4j2

major goal of the action plan is to improve vulnerability discovery and remediation. It proposes that developers be given facilitated access to advanced security tools. Until now, these tools have generally been too expensive due to licensing and server costs. At the same time, the plan calls for maintainers to be supported by paid and vetted security experts to assess potential security vulnerabilities and assist with remediation. Eric Brewer, Google VP of Infrastructure & Google Fellow, states that “We’re not too bad at finding defects, but we usually struggle to get them fixed." (CISA 2022) Thus, successful mitigation depends not only on the discovery of the vulnerability, but also the subsequent remediation and the successful delivery of updates to the end user.

One of the most advanced vulnerability scanners is CodeQL[3]. This static code analyzer was originally developed by Semmle before the company was acquired by GitHub in 2019[4]. In mid-2020, GitHub launched a native code scanning workflow that the user only has to activate once[5]. By default, CodeQL performs scans weekly, as well as scans on push and pull requests. However, the workflow can also be adapted to individual needs. The analysis is performed on GitHub’s cloud servers.

Since enabling CodeQL is free for public repositories, it levels the playing field between proprietary products and open source software. However, as already mentioned, the problems identified will eventually need to be fixed. This study therefore examines the efficiency and impact of CodeQL on repositories that have enabled this workflow.

[3]https://codeql.github.com/

[4]https://github.blog/2019-09-18-github-welcomes-semmle/

[5]https://github.blog/changelog/2020-05-06-github-advanced-security-code-scanning-now-available-in-limited-public-beta/

2.2 Research questions

This study examines two putative effects of CodeQL on a repository:

- Users identify and resolve security issues that were part of the code base before CodeQL was enabled. To confirm this effect, the number of problems should decrease after CodeQL is enabled.
- Users discover new security vulnerabilities early in development. Since a scan is performed by default for every change, developers are given the opportunity to revert problematic changes. To confirm this assumption, the trend line of average problems should flatten through CodeQL.

It is difficult to separate these two aspects. Therefore, the Research Question for this study is formulated somewhat more generally: **How efficient is CodeQL in increasing the code security of a repository?**

The further research question is: **What impact do the different workflow configurations have on the effectiveness of CodeQL?**

Efficient in this context means, how many of the issues raised by CodeQL are actually fixed. It does not refer to how technically mature CodeQL is, i.e. how accurately it can identify problems.

2.3 Related work

A large number of studies have already been conducted on various cyber security topics, such as classifying malware, developing algorithms to find vulnerabilities, detecting attacks early in their development, remediation, creating methods and tools to detect hidden attacks of new viruses. This study focuses on the effects of CodeQL that can be observed in real-life samples.

A slightly older study by Johnson et al. (2013) states that error messages from static analyzers are often ignored by developers and makes suggestions for improvement. According to the paper, the tools available at that time did not provide enough information on how a developer should handle a finding. Johnson et al. suggests that more informative messages as well as easy-to-apply fixes could improve effectiveness. They also propose that tools should integrate more seamlessly into a developer's existing workflow.

Avgustinov et al. (2015) presented a method to analyze individual developer coding habits. Their paper, "Tracking Static Analysis Violations Over Time to Capture Developer Characteristics", shows that developers can be assigned fingerprints based on

their interactions with violations. The contributions to four open source projects were taken into account, while users with less than 50 commits or 5,000 code line changes were excluded.

Bandara et al. (2021) used CodeQL to study the vulnerability lifecycle in 130 open source repositories. For this purpose, the JavaScript code in 501,000 commits of these repositories was examined. They found that 90% of the projects had a commit that was supposed to close a vulnerability but actually introduced at least one new vulnerability. Only 84% of the commits that were supposed to close a vulnerability did not create any new issues, while 18% of the total issues came from commits that were supposed to resolve a vulnerability but instead created new ones. This study also concludes that 78% of security vulnerabilities could have been avoided through proper internal testing.

In an earlier paper, Bandara et al. (2020) already reported comparable results. However, only a preliminary dataset with 118,000 commits from 53 repositories was analyzed.

3 Methods

3.1 Sample

A central issue of this paper was finding repositories that use CodeQL. Several approaches were used, which had different success rates, runtimes and result biases. In order to compare the repositories well, each project had to meet the following requirements:

- Repository uses JavaScript: This study focused exclusively on the analysis of JavaScript code, as it is very popular among developers[1] and is also used for both backend and frontend applications. Furthermore, no platform-dependent compilation or individual build-commands are necessary with JavaScript, which makes it easier to analyze projects on a large scale.

- Repository has at least one commit where a file was created or modified on the following path: .github/workflows/codeql-analysis.yml
  This is the default file name and location of the CodeQL configuration file. Theoretically, it would be possible that the developer chooses an alternative name when creating the workflow, but this is comparatively rare and prevents a performant and standardized decision as to whether the repository uses CodeQL.

- CodeQL has been continuously scanning the project's JavaScript code since activation.

- Repository had a commit after September 30, 2020. In this study, security is viewed as a process rather than an end state. In order to be able to analyze the effects of CodeQL, there must also be ongoing development. Projects that have not been further developed since September 30, 2020 were therefore declared as abandoned and excluded from the analysis.

[1] https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language

- Repository is not a fork. When a fork is created, it inherits the commit history of its parent. When snapshots are analyzed, they therefore reflect the state of the parent of the fork, skewing the results by giving increased weight to these duplicate commits.

Furthermore, some repositories had to be manually excluded from the analysis. Originally, the sample contained over 400 projects that obviously serve educational and demonstration purposes, some even with intentional security vulnerabilities. These include most notably copies of "OWASP Juice Shop"[2] and "WebGoat"[3]. Given the size of the sample, some repositories may have been overlooked, which could affect the results.

The evaluation also excluded the repository "respec" by w3c (GitHub-ID: 4287066). This project fixed 1,200 vulnerabilities about 5.5 years before CodeQL was enabled and shows no further change that can be attributed to CodeQL. Due to the very high number of 1,200 vulnerabilities, this single repository would strongly influence the average and thus distort the results.

The effort to check the requirements for a given GitHub repository with the GitHub API is manageable. However, GitHub limits API access to 5,000 calls per hour. To extrapolate the time required to compile a large perfectly randomized sample of repositories with a CodeQL action, 32,935 random, publicly available CodeQL repositories were examined. It was found that only 10 of these repositories met the requirements. In order to create a reasonable sized sample of repositories, the randomized crawler would have to run for several months.

The majority of the projects in the sample were crawled using the GitHub Search API. The results returned by the Search API cannot be considered complete, as GitHub's proprietary search algorithm makes certain trade-offs between completeness and performance.[4] The Search API provides up to 1,000 search results for a given search query, and this can be extended to up to 2,000 by sorting in descending and ascending order. The challenge was thus to select the search terms in such a way that the result does not exceed 2,000 hits, while at the same time not omitting any repositories.

Another challenge was the incomplete documentation of the API. In the end, the undocumented search parameter "committer-date:YYYY-MM-DD..YYYY-MM-DD" turned out to be helpful in identifying repositories with a high probability of having CodeQL enabled. By means of this parameter, all repositories were identified that had a commit in the period from May 1, 2020 to May 1, 2022, whose commit message indicated the activation of CodeQL. This means the commit messages contained certain keywords,

[2]https://github.com/juice-shop/juice-shop
[3]https://owasp.org/www-project-webgoat/
[4]https://docs.github.com/en/rest/search#search-code

for example: "codeql", "security", "scanning", "analysis", or "workflow".

Ultimately, over 2.5 million potential repositories were considered for the sample and screened for the requirements described above, of which 37,856 met them. Although such a wide range of repositories was considered, it is neither a complete list nor a perfectly randomized sample. After assembling a sufficiently large number of projects that use CodeQL, a random sample of 10,000 repositories was created from them. After completing the CodeQL scans, it became apparent that insufficient data had been collected for some repositories. In the subsequent analysis, therefore, only repositories for which data was available both in the 500-day period before CodeQL was activated and in the 200 days that followed were considered. For more information, see "Analysis of the Obtained Data and Data Cleansing".

3.2 Analyzing Snapshots

When analyzing projects, one can either examine all commits or just a selection of them. Previous studies analyzed repositories on a commit-by-commit basis, but in return, only a small number of projects were examined (Bandara et al. 2021). For this reason, a time-based selection of commits was made for this study. For each repository, commits were analyzed that were at least 30 days apart, starting from the most recent commit when capturing the commit data via GitHub API. 500 days before and 200 days after CodeQL activation, the interval was shortened to 7 days. To avoid bias effects due to interpolation in the days immediately before and after CodeQL activation, additional snapshots were analyzed during this period. This approach makes it possible to examine the vulnerability remediation progress of a large number of projects over time.

Depending on when the project was created and how many commits were made in the past, the number of analyzed commits per repository differs. In the evaluation of the data, interpolation is performed between the missing points, which allows a statement to be made about the average number of security vulnerabilities expected in the project at any point in time. It should be noted, however, that there may be short-term effects that have not been taken into account. If one or more security vulnerabilities are introduced in a project and patched just a short time later, there is a high probability that this process will not be detected by the rather coarse-meshed check every couple of days.

Once the commits were identified for analysis, a snapshot of the repository at that time was taken, downloaded from GitHub, and the CodeQL database for the JavaScript programming language was created. The query suite for CWE-79 vulnerabilities was then performed on this database.

This query suite contains the following rules:[5]

- Unsafe HTML constructed from library input: This rule applies when a library function dynamically constructs HTML in a potentially unsafe way without proper documentation.
- Reflected cross-site scripting: User input is returned to the user without properly sanitizing or escaping it. These vulnerabilities can be exploited, for example, to inject arbitrary JavaScript code into a trusted environment. Artifacts of this type are classified as errors by CodeQL.
- Stored cross-site scripting: The server generates a response from data stored on the system without appropriate sanitation or escaping.
- Unsafe jQuery plugin: The developer has written a jQuery plugin that has an insecure function that dynamically creates HTML without sufficient documentation.
- Client-side cross-site scripting: User input is displayed on the web page without sufficient filtering or sanitization.
- DOM text reinterpreted as HTML: Text is extracted from a DOM node and then interpreted as HTML.
- Exception text reinterpreted as HTML: An error message is displayed directly to the user, and parts of the error message could be manipulated and reinterpreted as HTML.

These rules can detect problems with a high precision (with possible values from low, medium, high to very-high)[6]. Thus, when interpreting the results, generalized statements cannot necessarily adequately account for the actual circumstances present in a given repository. On a scale of 0 (no vulnerability) to 10 (extremely critical), vulnerabilities found via the above rules were each rated 6.1 and are thus of medium severity. This value reflects the 75th percentile of CVSS scores assigned to CVEs matching this query.[7]

[5] https://github.com/github/codeql/tree/main/javascript/ql/src/Security/CWE-079
[6] https://codeql.github.com/docs/writing-codeql-queries/metadata-for-codeql-queries/
[7] https://github.blog/changelog/2021-07-19-codeql-code-scanning-new-severity-levels-for-security-alerts/

3.3 Analysis of the Obtained Data and Data Cleansing

A variety of metadata was collected for each repository, especially regarding the configuration of the CodeQL workflow and changes to it. In the following analysis of the results, the repositories can thus be compared with each other based on different attributes.

To quantify the impact of CodeQL activation, the tool CausalImpact[8] was chosen. It estimates the causal effect of the intervention by applying a Bayesian structural time-series model. Using a set of control time series, the expected vulnerability development is modeled both before and after workflow activation, making it possible to estimate the impact of CodeQL even without A/B testing.

Multiple control time series from the various CodeQL queries were used. Their data is based on 20,614 successfully analyzed snapshots from 725 repositories that also use CodeQL and scan other programming languages, but not JavaScript. The assumption is made that these projects place a similar emphasis on safety as the other projects, but do not experience a causal effect on their JavaScript parts from CodeQL.

The analysis only contains projects where at least one commit could be successfully analyzed in the time period of 500 days before as well as 200 days after CodeQL was activated. This limitation is necessary to quantify the impact of CodeQL activation in a meaningful way. In total this includes 8,931 repositories from 7,639 different GitHub users. The graphs in this study are based on 206,246 successfully analyzed snapshots from these projects. The analysis failed for 13,583 snapshots. Possible reasons why the analysis of a snapshot fails include that no valid JavaScript code was found or the corresponding commit was no longer publicly available at the time of the analysis. In addition, for some snapshots, TypeScript and thus Node.js was mandatorily required by CodeQL to perform the analysis. Because Node.js was not available on the computing cluster used to perform the CodeQL scans, these snapshots could not be successfully examined.

The time period in which successfully analyzed snapshots are available varies greatly depending on the repository. This is due to the different age and development progress of the individual projects. On average, the oldest snapshot is 570 days before and the most recent snapshot is 150 days after CodeQL was activated.

To perform an analysis with CodeQL, the evolution of vulnerabilities is first evaluated for each project and missing data is interpolated. Then, the average per day is calculated across all repositories.

[8] https://google.github.io/CausalImpact/CausalImpact.html

3.4 Limitations of This Study

This study focuses on JavaScript, ignoring other programming languages that CodeQL supports, such as Python, C++, Java or TypeScript. In addition, only security vulnerabilities of type CWE-79 were examined. Also, not every commit in a repository was examined. It is therefore conceivable that outliers falsify the results. However, due to the large number of repositories and snapshots examined, this effect is manageable.

Furthermore, the results of CodeQL cannot be equated with the true state of a repository, because false positives and negatives are inevitable. In addition, this study does not identify why developers do not fix a problem. Possible reasons, such as a lack of knowledge, skill or time, are not clear from the pure observation of the defects. In addition, developers may check issues indicated by CodeQL and, using their domain knowledge, conclude that it is not applicable and safely ignore it. It is likely that some of the problems found are only in the test suite and thus do not represent a real threat or are even purposeful.

Some repositories may choose to use additional automated code reviewing tools and services, such as Amazon CodeGuru[9], Code Climate[10], SonarQube[11] or SonarCloud[12]. Due to the structure of this study, interventions like these cannot be considered.

[9] https://aws.amazon.com/codeguru/
[10] https://codeclimate.com/
[11] https://www.sonarqube.org/
[12] https://sonarcloud.io/

4 Results

4.1 Results Across all Projects

4.1.1 All Rules Combined

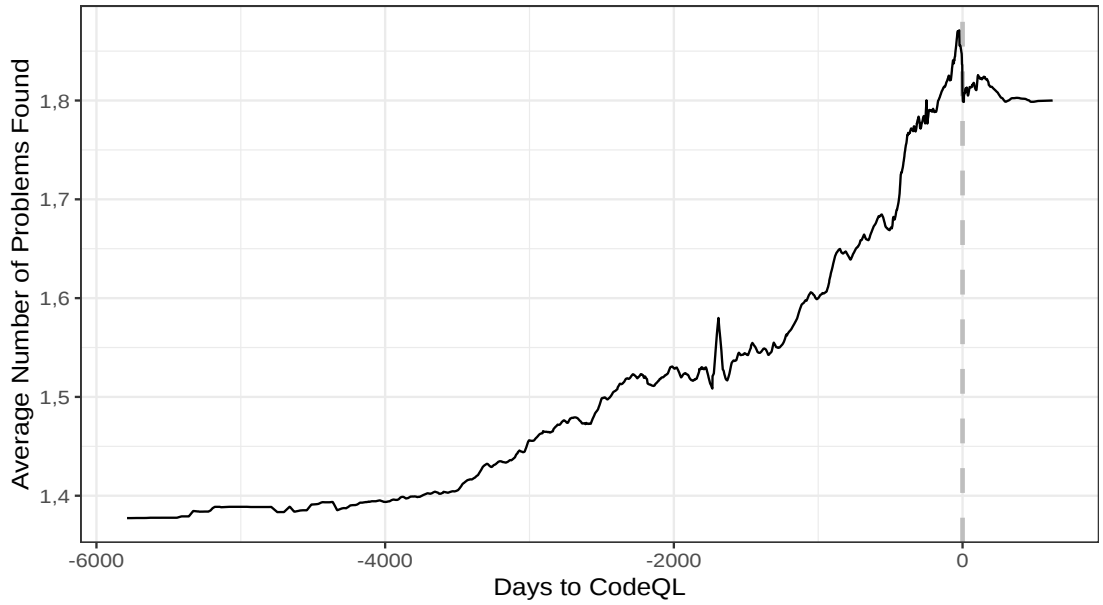

Figure 4.1: Evolution of vulnerabilities in the analyzed repositories in the period of about 5,788 days before CodeQL activation and 624 days after. Please note that more interpolation was needed in the marginal areas, as many repositories did not have any commits in this period, as they are still too young.

Repositories tend to accumulate security vulnerabilities over time (Figure 4.1). **The average number of findings in a repository after activation of CodeQL was significantly reduced.** Right after activation (marked by dashed gray line) the observed effect is greatest, but the positive trend lasts permanently. Please note, in this and also in most of the following graphs, the y-scale has been shifted upwards, making trends

easier to recognize, but the relative change may unconsciously be overestimated. The positive effect of CodeQL is particularly clear when the expected development without intervention is calculated based on a control time series that helps to quantify the causal effect of CodeQL (Figure 4.3).

As the size of JavaScript code in a project increases, the average number of problems found tends to increase as well (Figure 4.2). However, the large scatter also means that this effect should not be overstated. Considering the hypothesis that increasing complexity is a decisive factor for more bugs, this means that the size of the code cannot be equated with its underlying complexity.

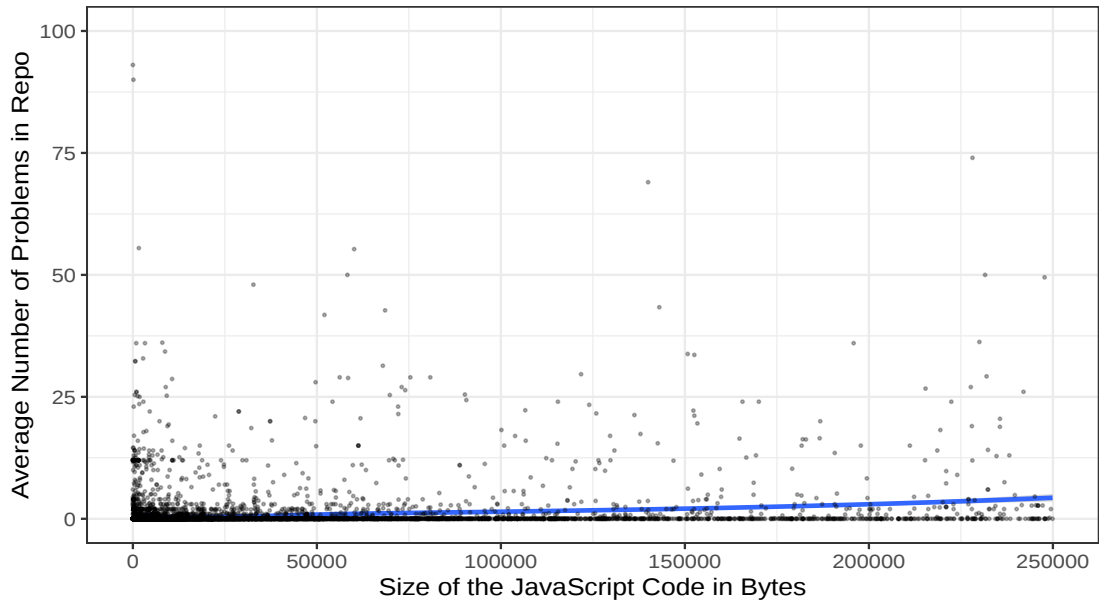

Figure 4.2: The average number of issues found in a repository vs the size of the JavaScript code in the project (some points are out of scale). Each point corresponds to a repository from the sample. The blue line indicates the local regression.

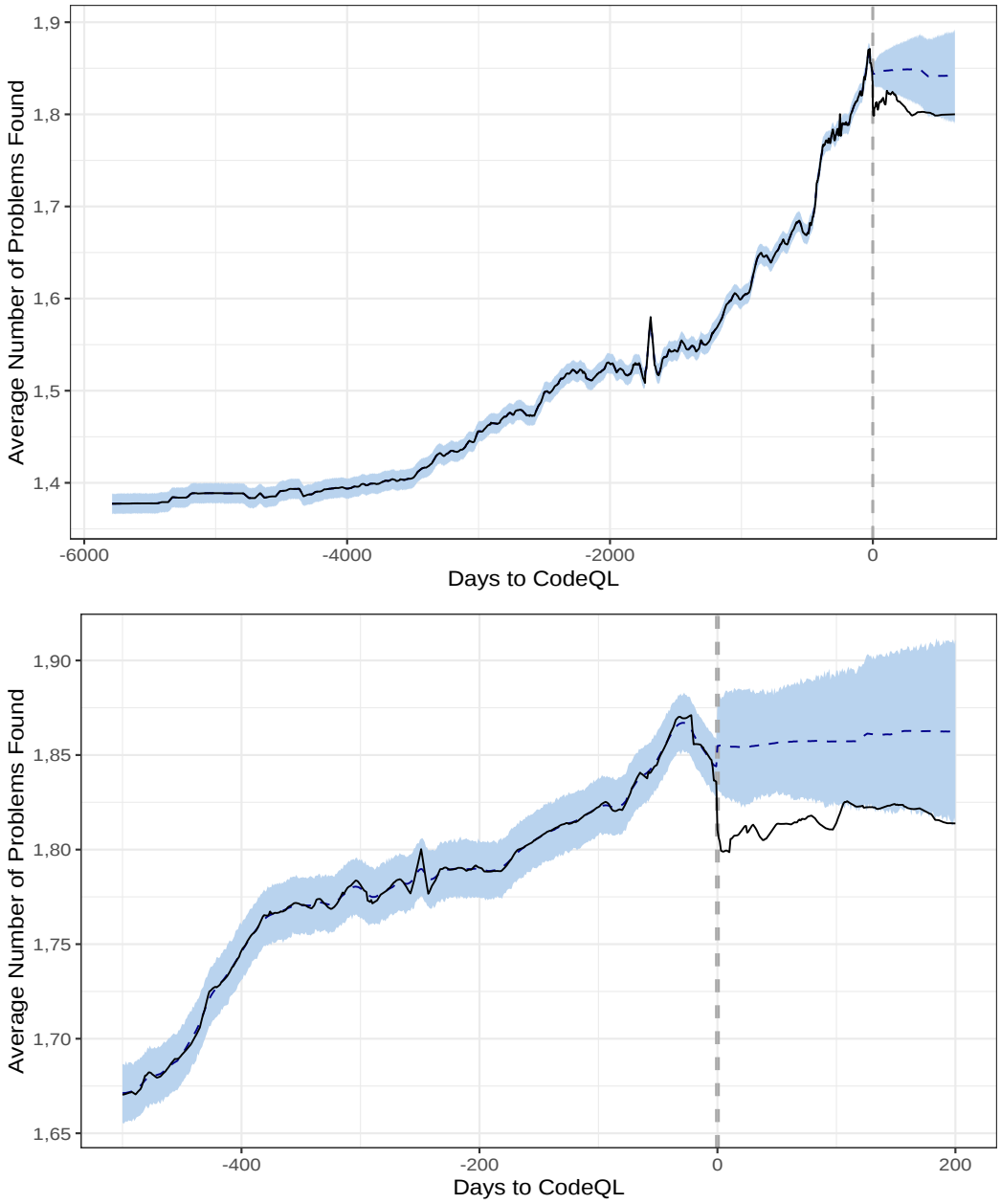

Figure 4.3: CausalImpact analysis of the combined development of all rules under study. The upper graph covers 5,788 days before as well as 624 days after activation of CodeQL. The second graph covers 500 days before as well as 200 days after activation of CodeQL. The dashed line shows the development hypothesized by CausalImpact. The area highlighted in blue indicates the 95% interval.

During the post-activation period, CodeQL detected an average of 1.81 vulnerabilities. Without the intervention, however, we would have expected an average value of 1.85, with a 95% confidence interval of [1.82, 1.87]. This means that maintainers of a repository fixed an average of 0.039 vulnerabilities (95% confidence interval: [0.011, 0.067]) that CodeQL alerted them about. This corresponds to a 2% reduction in security vulnerabilities, with a 95% confidence interval of [1%, 4%]. The posterior probability of a causal effect of CodeQL is 99.2%. The Bayesian one-sided tail-area probability of observing this effect by chance is p = 0.008. Thus, the causal effect of CodeQL can be considered statistically significant.

The downward pointing trend line (Figure 4.4) after activation confirms that maintainers with CodeQL enabled tend to add fewer vulnerabilities to the code.

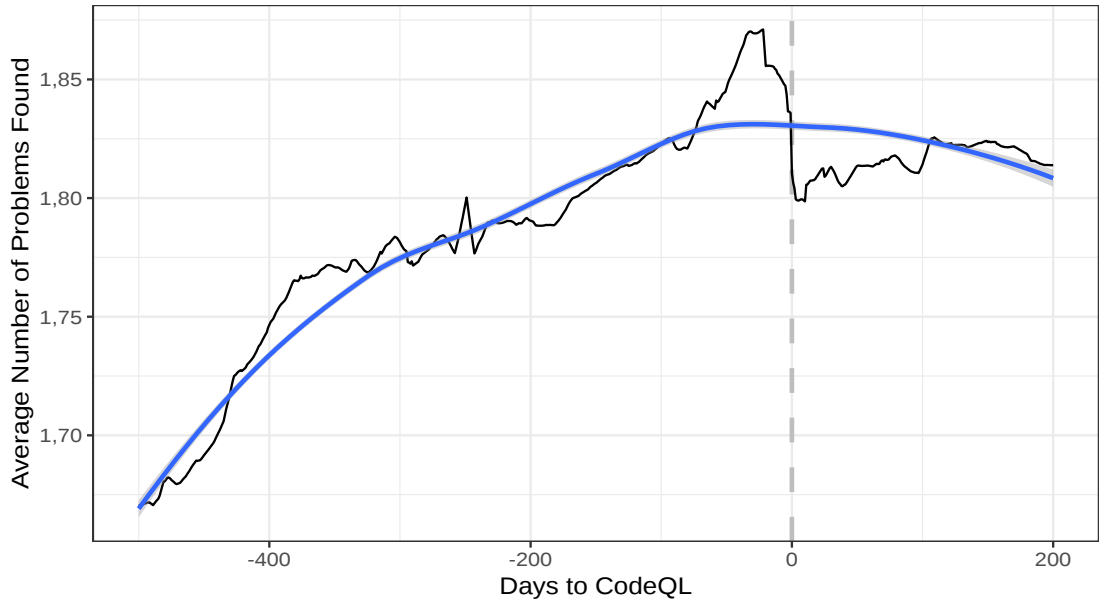

Figure 4.4: Development of the average number of defects. The local regression is plotted in blue, the 95% interval is highlighted in gray.

The causal effect of CodeQL varies enormously between different repositories. The following numbers are based on the average of each repository in the 45-day period before activation compared to 45 days after. 102 projects fixed 95% or more of the issues identified by CodeQL while 152 projects at least improved. No change was detected in 1,077 of the repositories that had a vulnerability alert at that time. 262 repositories added new security vulnerabilities (Figure 4.5). This means that around 67.6% of repositories did not improve despite CodeQL alerts, while 16.4% actually

worsened. Only 15.9% of the repositories with problems improved. Nevertheless, on average, the repositories improved.

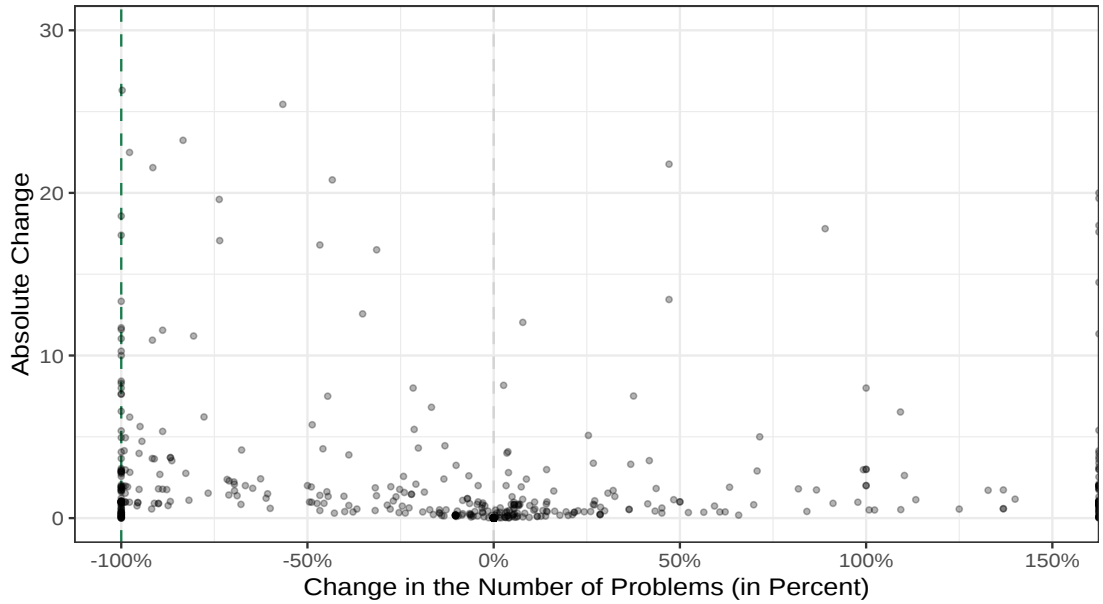

Figure 4.5: The percentage change in the average number of problems found in a repository over the 45-day period before and after CodeQL was activated vs the absolute change. Each point corresponds to a repository from the sample. A point at zero means there was no change, a point is on the green dashed line when all issues were fixed. The higher a point is, the more vulnerabilities were removed or added. Some points are out of scale.

Of the 8,931 repositories examined, no CWE-79 vulnerabilities could be identified in 7,089 (79.4%) of them. At least one vulnerability was found in 1,842 (20.6%) projects. Most of the findings, 46.1% on average, were found via the DOM Text Reinterpreted as HTML rule. Insecure jQuery plugins are responsible for 32.2% of issues. 13.5% problems contained code that constructs HTML from input, while 6.0% of issues are XSS defects. 1.1% of issue are due to possible Reflected XSS attacks and 0.8% of problems exposed XSS through Exception vulnerabilities. Only 0.2% of defects fall into the Stored XSS category. The absolute biggest change of CodeQL can be seen in DOM Text Reinterpreted as HTML (Figure 4.6 & 4.6).

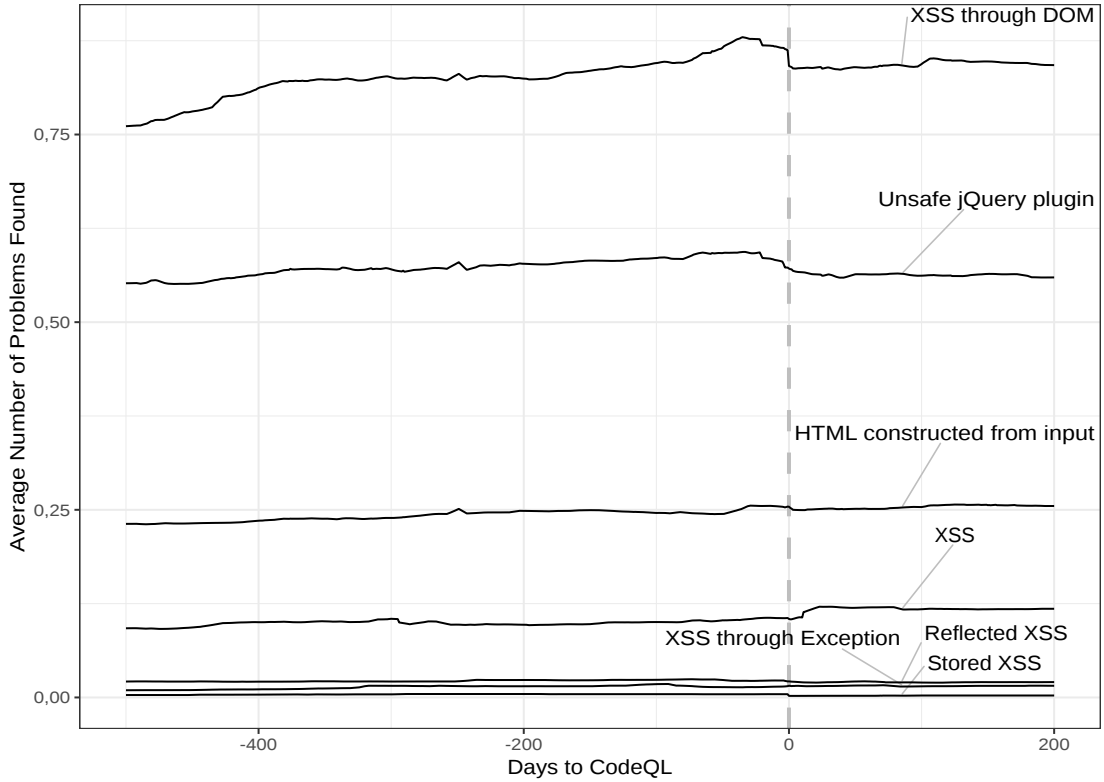

Figure 4.6: The plot shows the evolution of the average problems found in a repository split by rules.

4.1.2 DOM Text Reinterpreted as HTML (XSS through DOM)

After activation, an average of 0.84 issues of the type DOM Text Reinterpreted as HTML were identified. However, without CodeQL, we would have expected an average value of 0.87, with an 95% confidence interval of [0.85, 0.88] for this counterfactual prediction (Figure 4.8). Relatively speaking, this corresponds to an improvement of 3% (95% interval: [2%, 5%]). In absolute terms, an average of 0.028 vulnerabilities were patched, with an 95% interval of [0.014, 0.042]. The causal effect of CodeQL can be considered statistically significant, since the Bayesian one-sided tail-area probability of obtaining this effect by chance is very small with p = 0.001. Immediately after CodeQL is enabled, there is a significant drop in vulnerabilities. However, the downward trend is not so significant in the later course. This suggests that an alert about a security vulnerability that is not immediately fixed is likely to be ignored even in the longer term. At the

same time, however, the stagnation also contrasts with the previous observation that problems increase over time. Thus, CodeQL presumably also prevents problems from being added to the code base and persisting.

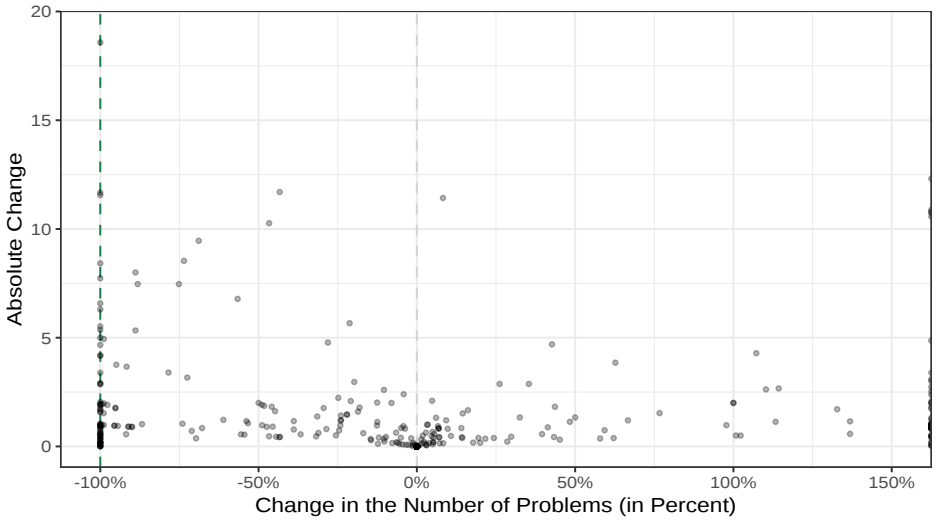

Figure 4.7: The percentage change in the average number of "DOM Text Reinterpreted as HTML" issues found in a repository over the 45-day period before and after CodeQL was activated vs the absolute change.

Looking at the development for the repositories individually, the results are mixed (Figure 4.7). 1,146 repositories were found to have no problems in the 45-day period before and after CodeQL was activated. However, 822 repositories (71.7%) with problems showed no impact from CodeQL, while 149 (13.0%) actually worsened in the post-activation period. 77 repositories (6.7%) removed at least 95% of the problems, and another 98 repositories (8.6%) showed improvement.

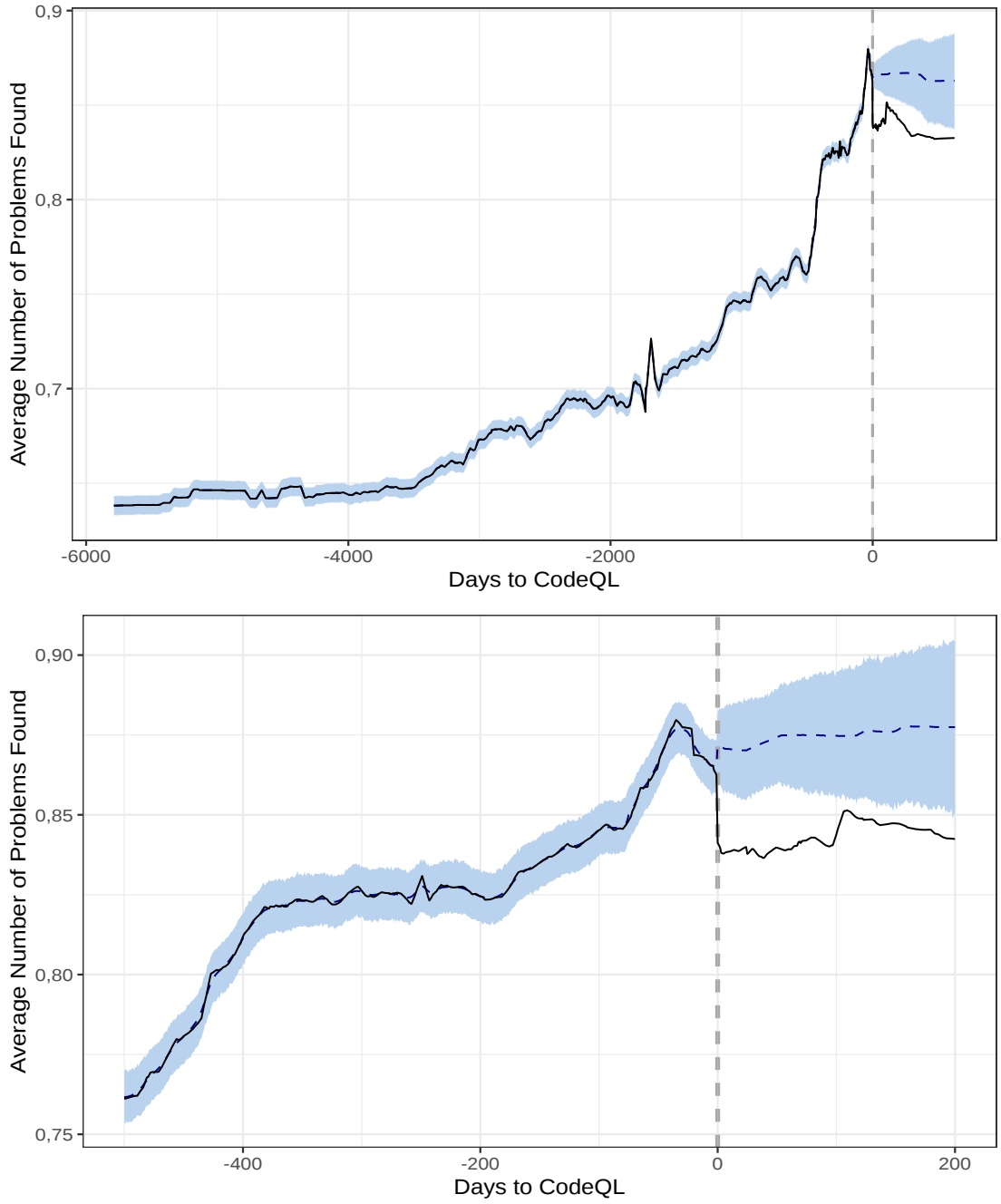

Figure 4.8: CausalImpact analysis of the rule "DOM Text Reinterpreted as HTML"

4.1.3 Unsafe jQuery Plugin

The average number of insecure jQuery plugins decreased from an expected 0.58 (95% interval: [0.57, 0.59]) to 0.56 (Figure 4.9). Thus, an average of 0.023 vulnerabilities were fixed, with a 95% interval of [0.013, 0.032]. This corresponds to an average reduction of 4%, the 95% interval for this being [2%, 6%].

The probability of observing this effect by chance is very small; the posterior probability of a causal effect is 99.9%. Therefore, the effect can be interpreted as statistically significant.

Of the 578 repositories that had an unsafe jQuery plugin defect in the 45-day period before and after CodeQL was activated, no change was detected in 456 (78.9%) of them. Only 29 (5.0%) removed 95% or more of the issues and another 32 (5.5%) also improved, while 61 projects (10.6%) showed worse performance (Figure 4.10).

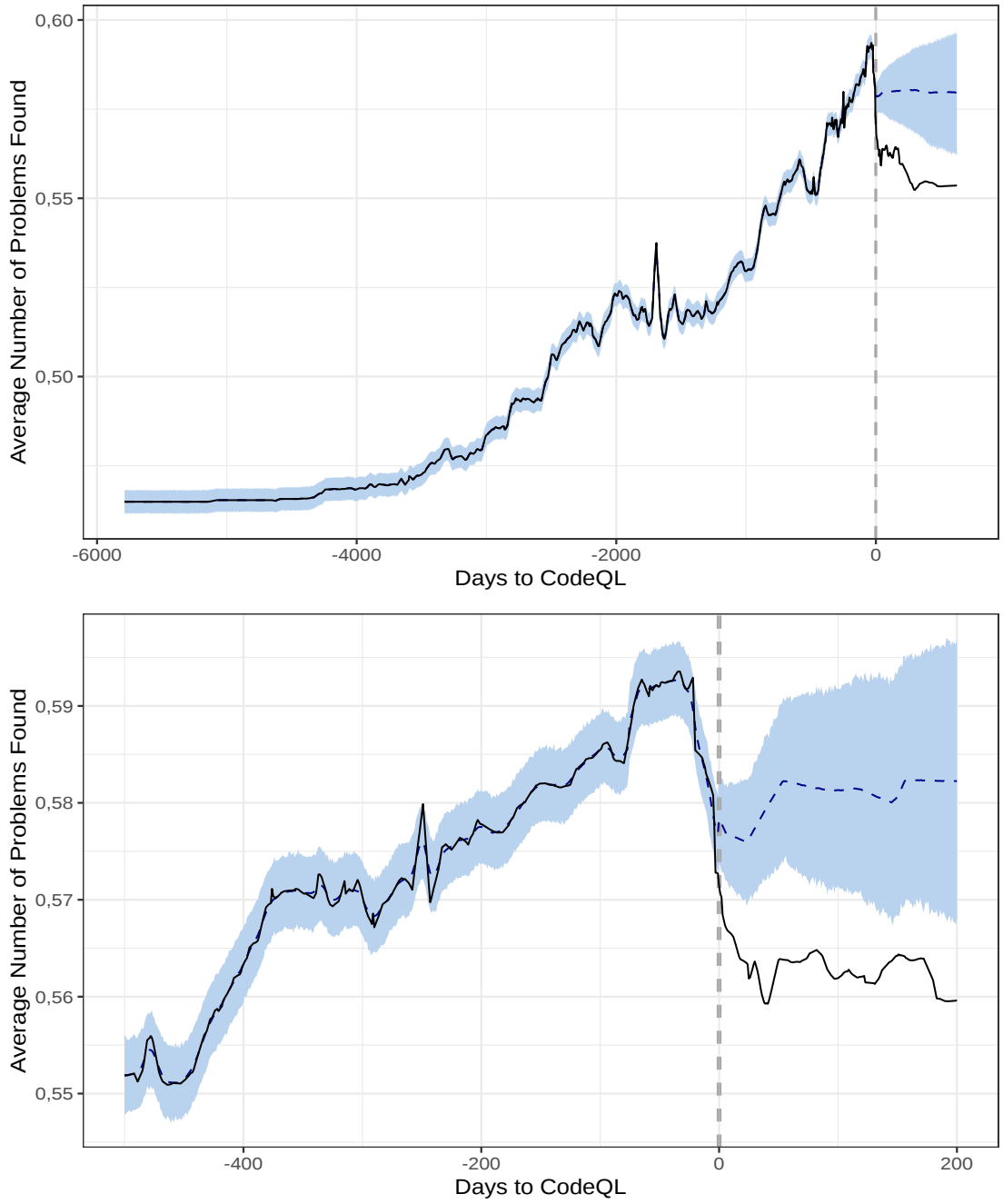

Figure 4.9: CausalImpact analysis of the rule "Unsafe jQuery Plugin"

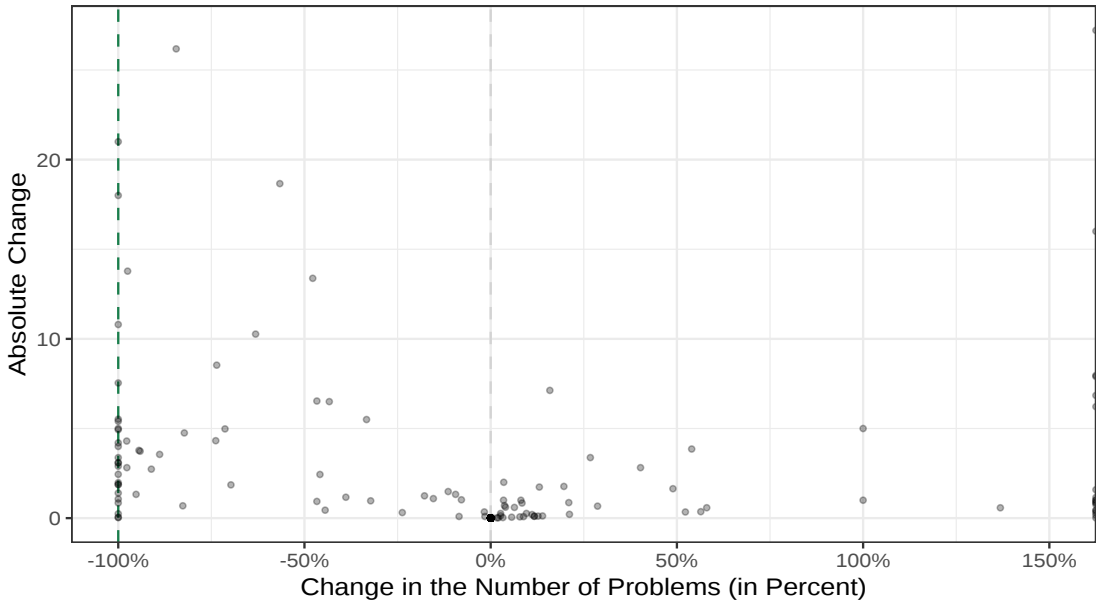

Figure 4.10: The percentage change in the average number of "Unsafe jQuery Plugins" issues found in a repository over the 45-day period before and after CodeQL was activated vs the absolute change.

4.1.4 Unsafe HTML Constructed From Library Input

The impact of CodeQL on problems of type Unsafe HTML Constructed From Library Input is unspecific (Figure 4.11 & 4.13). In the post-intervention period, an average of 0.26 errors were detected, with the expected number being 0.25 (95% interval: [0.25, 0.26]). Problems increased by an average of 0.0013, with a 95% interval of [-0.0041, 0.0069]. The effect cannot be considered statistically significant because the chance of observing it only by chance is 30.3%.

Looking at the difference between the observed data and the counterfactual prediction (Figure 4.13), it can be seen that there is a noticeable decline of problems in the first few days after CodeQL is activated. It appears that the effect may be short-lived or that unrelated effects are skewing the data.

233 repositories in the sample showed a change in the number of Unsafe HTML Constructed From Library Input issues. Thus, more data and a larger sample would be needed to better investigate the effect.

In the 45-day period before and after CodeQL was enabled, issues were identified in 255 repositories. 192 projects (75.3%) showed no change from before to after, 25 repositories (9.8%) removed more than 95% of the problems, another 14 (5.5%) improved, and 24 projects (9.4%) worsened their performance.

This again shows that the effect of CodeQL varies greatly depending on the repository.

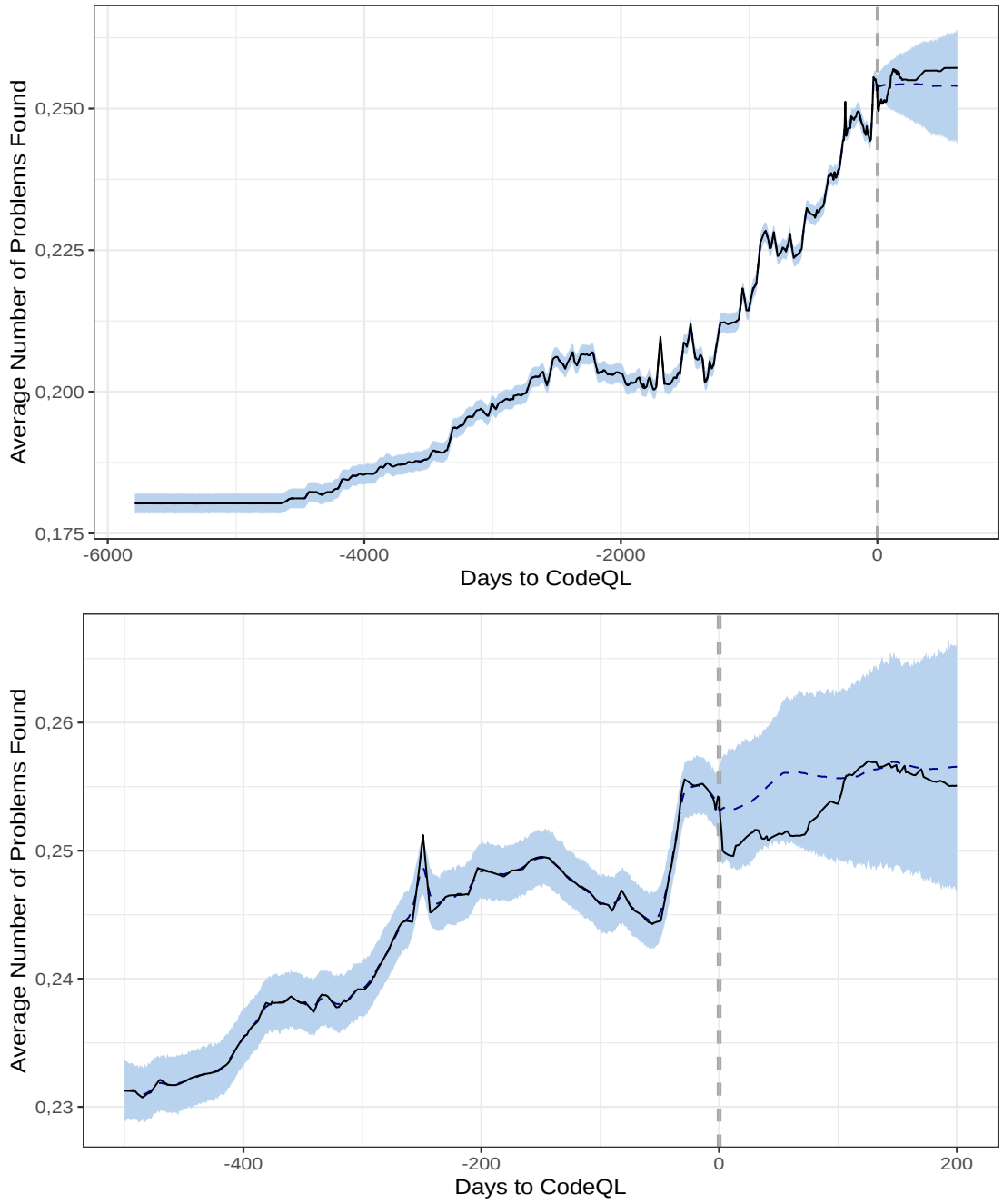

Figure 4.11: CausalImpact analysis of the rule "Unsafe HTML Constructed From Library Input"

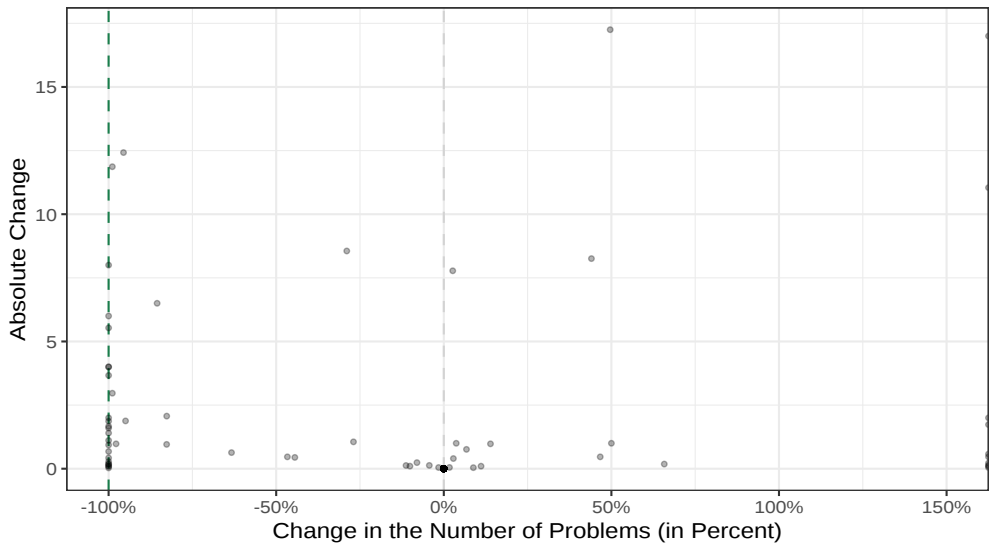

Figure 4.12: The percentage change in the average number of "Unsafe HTML Constructed From Library Input" issues found in a repository over the 45-day period before and after CodeQL was activated vs the absolute change.

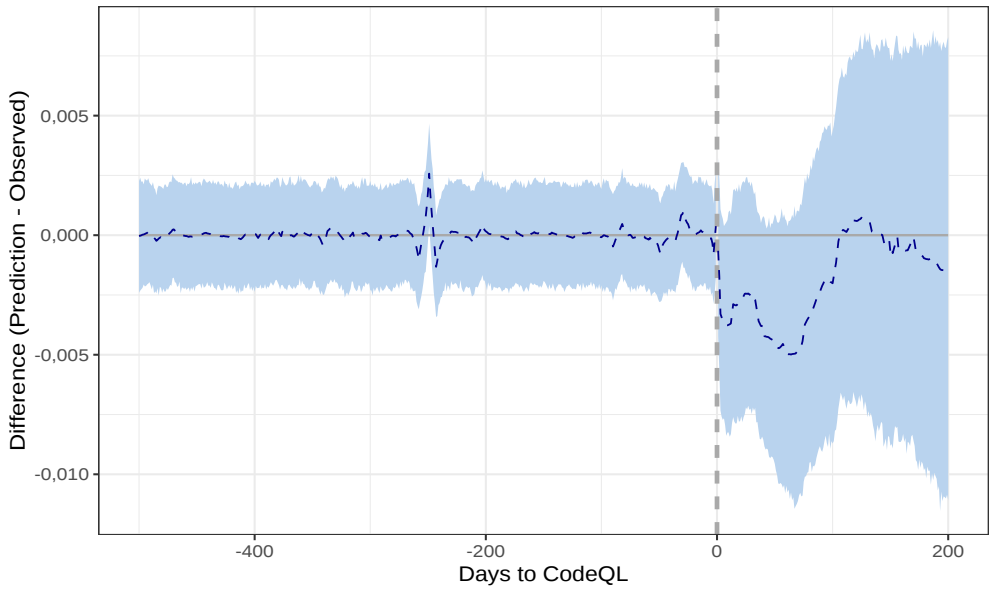

Figure 4.13: Difference the measured number of problems and counterfactual prediction by CausalImpact.

4.1.5 Client-Side Cross-Site Scripting

Shortly after CodeQL is enabled, the average number of Client-Side Cross-Site Scripting vulnerabilities increases sharply (Figure 4.14). This is in strong contrast to the expectation that CodeQL will reduce the number of security issues. In the period with CodeQL enabled, the average number of problems was 0.12; without CodeQL, we would have assumed an average number of 0.11, with a 95% interval of [0.10, 0.11].

Thus, the average number of findings increased by 0.012, with a 95% interval of [0.0092, 0.015]. These findings are also statistically significant, since the chance of a causal effect is 99.9%.

On the repository level, it is apparent that these results are due to two outliers (Figure 4.15).

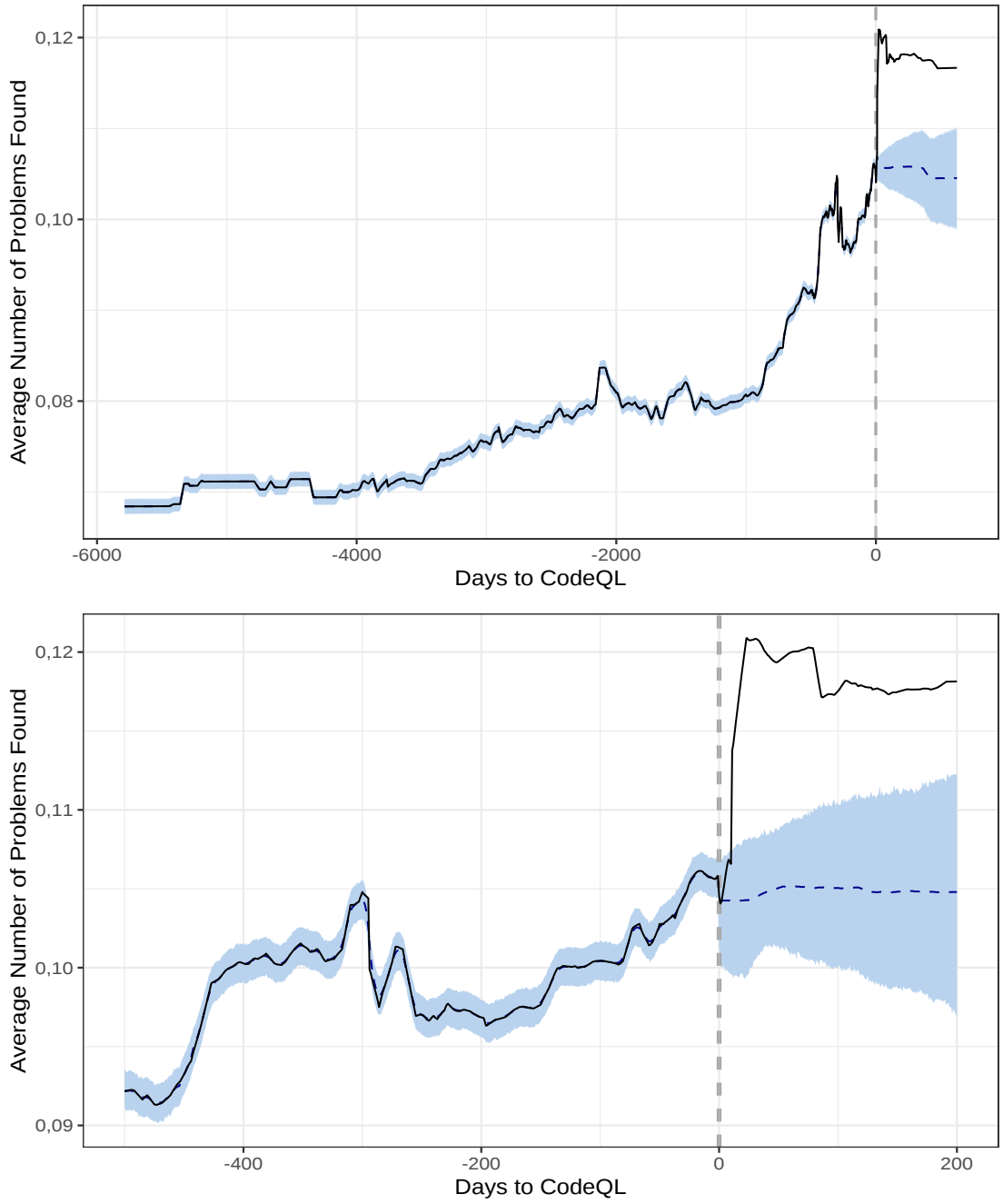

Figure 4.14: CausalImpact analysis of the rule “Client-Side Cross-Site Scripting”

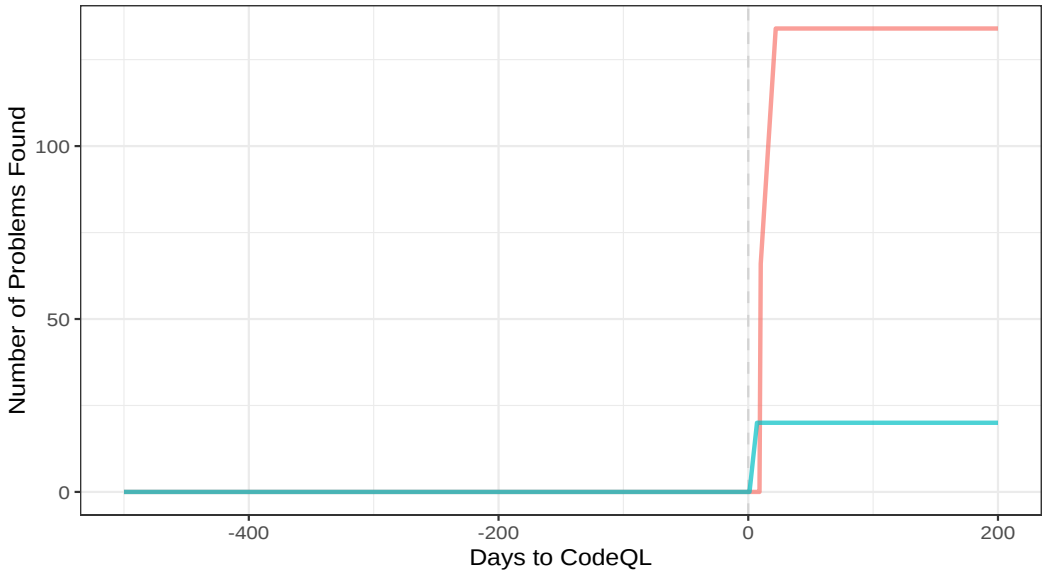

Figure 4.15: Development of XSS vulnerabilities in the repository "ComputerEmulator" owned by the user smorgo (GitHub Repository-ID: 334191697, in red) and "do-it-example" by lama-org (GitHub Repository-ID: 369692262, in turquoise)

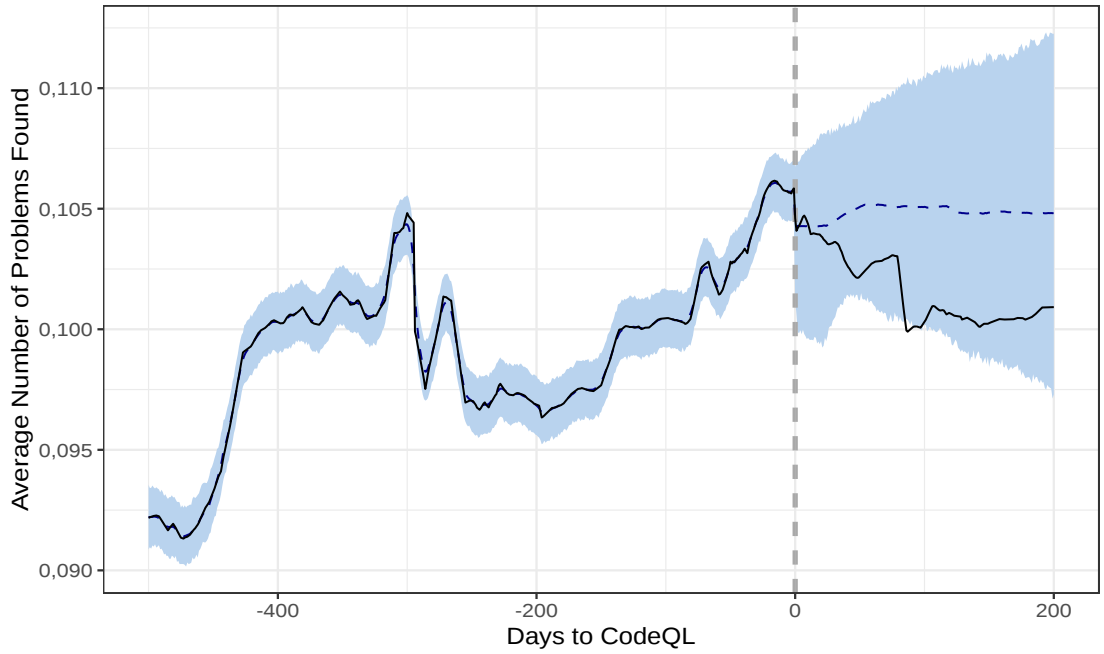

Figure 4.16: CausalImpact analysis of the rule "Client-Side Cross-Site Scripting", excluding the repositories "ComputerEmulator" by smorgo (GitHub Repository-ID: 334191697) and "do-it-example" by lama-org (GitHub Repository-ID: 369692262))

When excluding both of these repositories, the average number of problems after activation of CodeQL is 0.10, with the counterfactual prediction remaining 0.11 (95% interval: [0.10, 0.11]) (Figure 4.16). Thus, the effect is -0.0047 with a 95% interval of [-0.0078, -0.0016]. This corresponds to a reduction of -4%, the 95% interval being [-7%, -1%]. The probability of a causal effect is 99.3%, and the Bayesian one-sided tail-area probability of obtaining it by chance is 0.7%, which means that the effect can be considered statistically significant.

If one examines the effects on the period of 45 days before and after activation of CodeQL, a similar picture emerges as already with the other CodeQL queries: Some repositories improve, while a large portion seem to show no effect from CodeQL (Figure 4.17).

25 repositories resolved 95% or more of the problems found. Another 40 repositories also improved, while 70 repositories added new vulnerabilities to the codebase. 252 repositories had an XSS issue and did not change in this 90 day period.

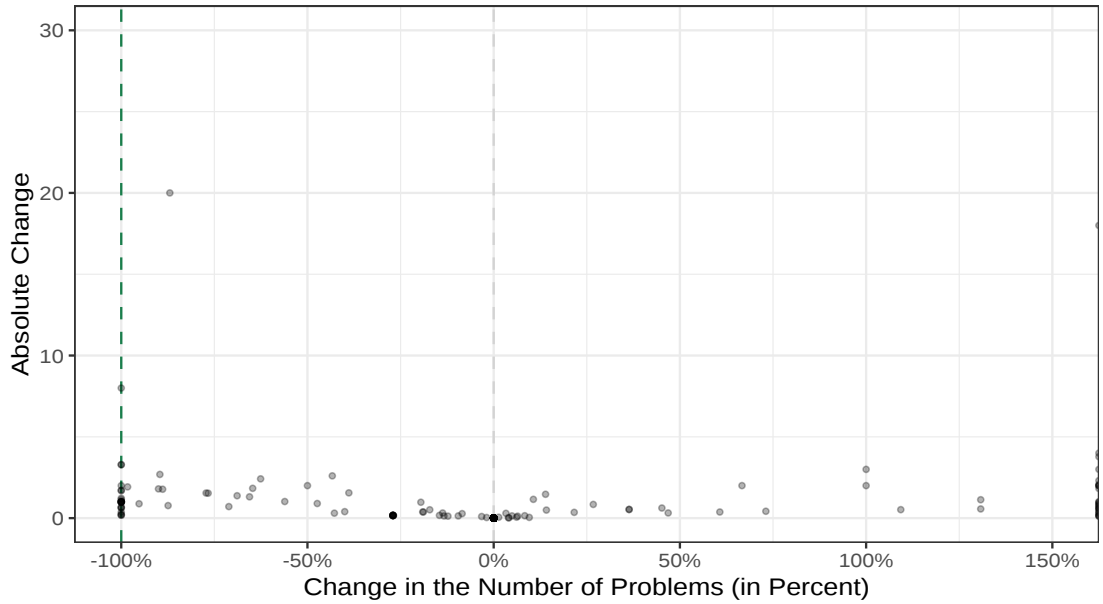

Figure 4.17: The percentage change in the average number of "Client-Side Cross-Site Scripting" issues found in a repository over the 45-day period before and after CodeQL was activated vs the absolute number of added or fixed vulnerabilities.

4.1.6 Reflected XSS

In the period with CodeQL-scanning enabled, an average of 0.021 defects of the Reflected XSS type were found (Figure 4.18). Without the effect of CodeQL, however, we would have expected an average of 0.022 problems, with a 95% interval of [0.022, 0.023]. Thus, an average of 0.0014 problems were fixed, which corresponds to a reduction of 6%. The 95% confidence intervals are [0.00063, 0.0022] and [3%, 10%]. Since the probability of obtaining this reduction by chance is very small (Bayesian one-sided tail-area probability: 0.1%), the effect can be considered statistically significant.

Of the 112 repositories where issues were identified in the 45-day period before and after CodeQL activation, 56.3% (63 projects) showed no change, 14 projects (12.5%) removed more than 95% of the problems, another 12 (10.7%) improved, and 23 projects (20.5%) saw their performance deteriorate (Figure 4.19).

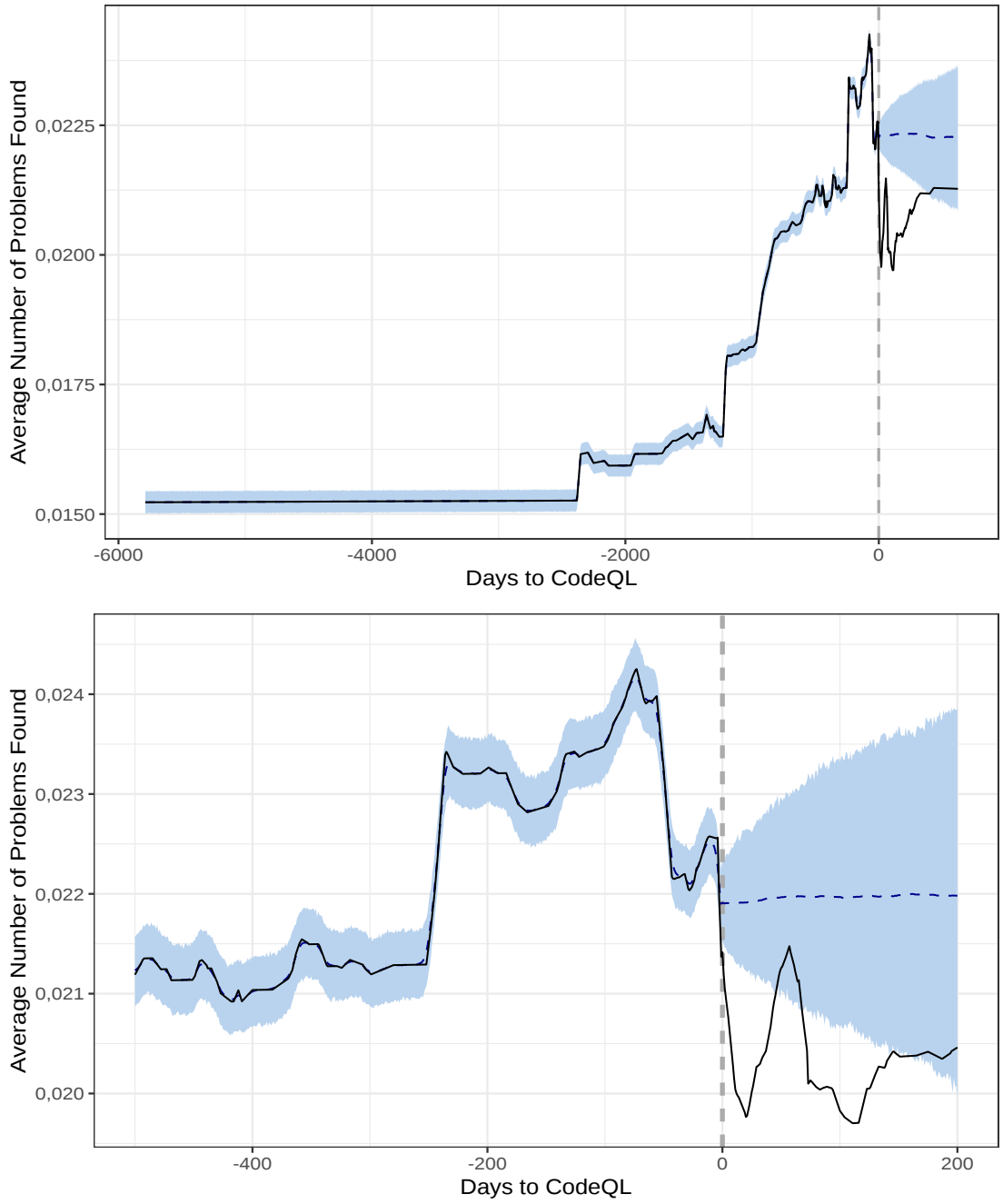

Figure 4.18: CausalImpact analysis of the rule “Reflected XSS”

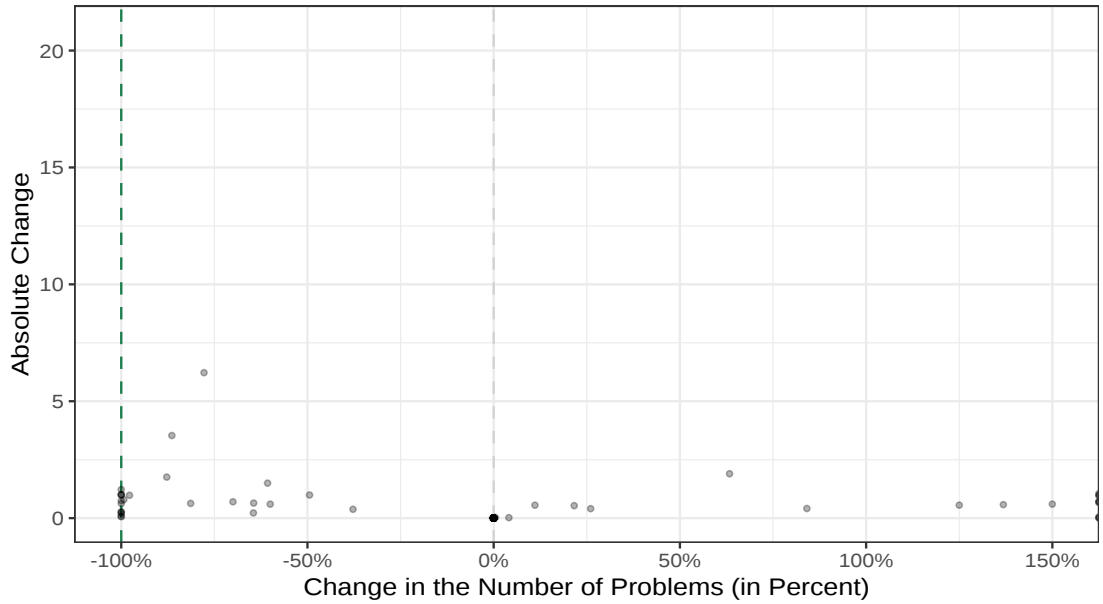

Figure 4.19: The percentage change in the average number of "Reflected XSS" issues found in a repository over the 45-day period before and after CodeQL was activated vs the absolute number of added or fixed vulnerabilities of that type.

4.1.7 XSS through Exception

57 repositories out of the 8,931 repositories examined contained XSS through Exception vulnerabilities in the 45-day period before and after CodeQL was activated. Since the statistics are based on only a small number of projects, the results are not as reliable and conclusive compared to the previously examined queries.

With CodeQL activated, an average of approximately 0.016 problems were identified, and the expectation without the intervention was 0.015 with a 95% interval of [0.014, 0.016] (Figure 4.20). Therefore, the number of problems increased by 0.0011 or +8% with a 95% interval of [0.00013, 0.0021] and [1%, 14%]. This effect can be considered statistically significant because the probability of observing it by chance is 1.1%.

Of the 57 projects that showed problems in the 45 days before or after CodeQL was activated, 10 repositories improved, 8 of which improved by more than 95%. For a little more than half of the repositories, 29 projects, the number of findings did not change, while for 18 repositories (32.1%) the number of issues increased (Figure 4.21).

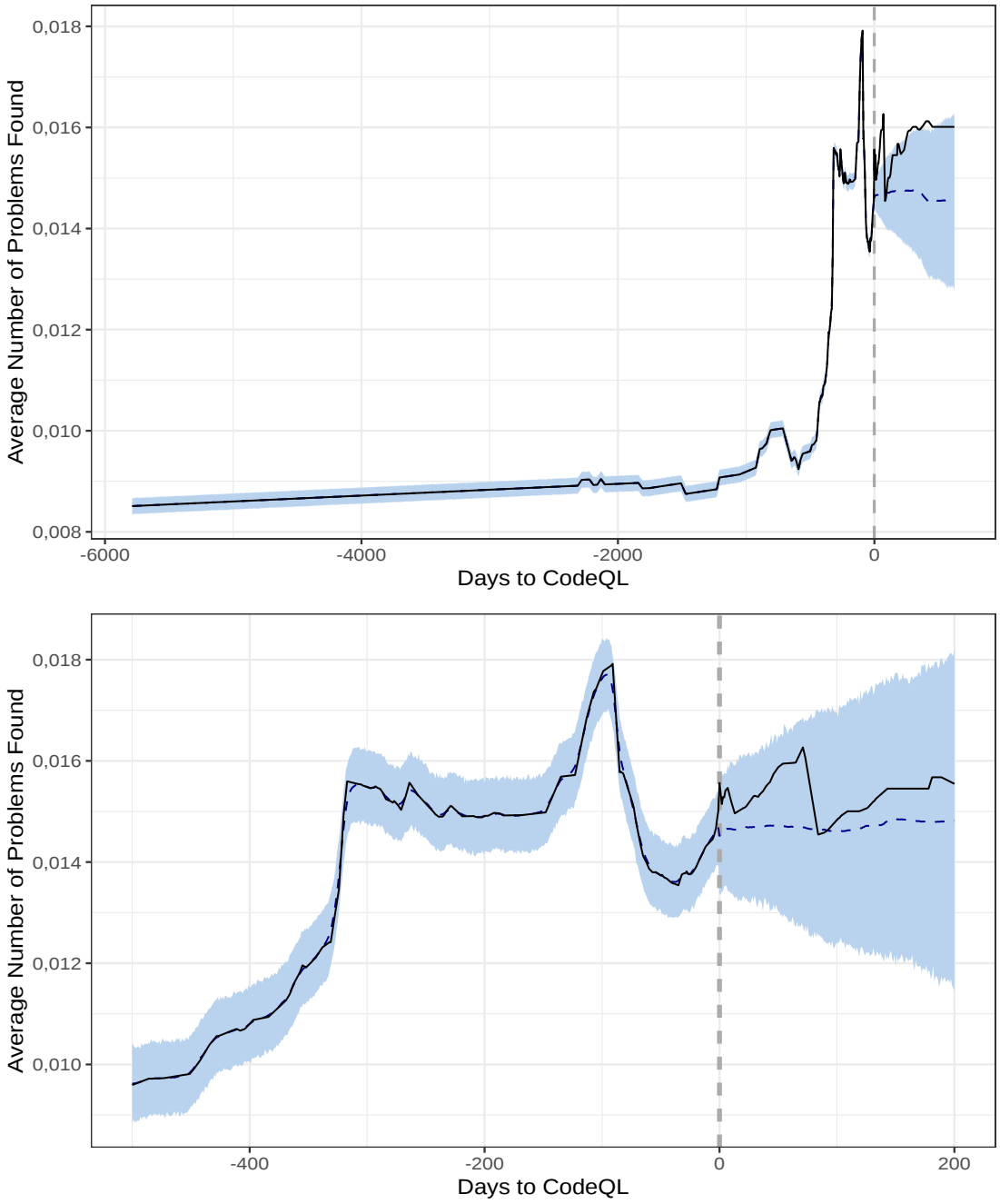

Figure 4.20: CausalImpact analysis of the rule "XSS through Exception"

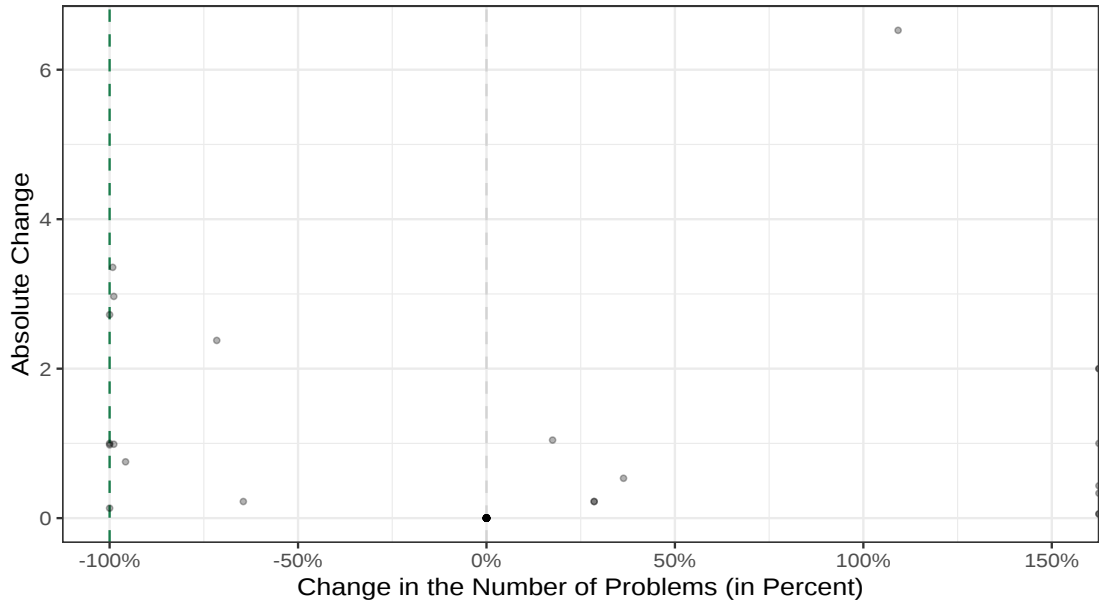

Figure 4.21: The percentage change in the average number of "Reflected XSS" issues found in a repository over the 45-day period before and after CodeQL was activated vs the absolute number of added or fixed vulnerabilities of that type.

4.1.8 Stored XSS

Only 11 repositories from the sample showed a change in stored XSS vulnerabilities. Thus, the sample was too small to include a sufficiently large number of projects with relevant findings.

To study the effects of CodeQL, the period just before and after activation is most interesting. During these 90 days, 6 repositories were found to have problems. Of these, 3 projects showed no change, one project improved by 100%, and two repositories worsened.

The CausalImpact analysis shows that the number of problems decreased by 40% after CodeQL, with a 95% interval of [35%, 45%] (Figure 4.23). This effect is exclusively due to the repository ”livepraise” by cadimos (GitHub Repository-ID: 305507071). The maintainer removed all 20 Stored XSS vulnerabilities within a short time after he activated CodeQL (Figure 4.22).

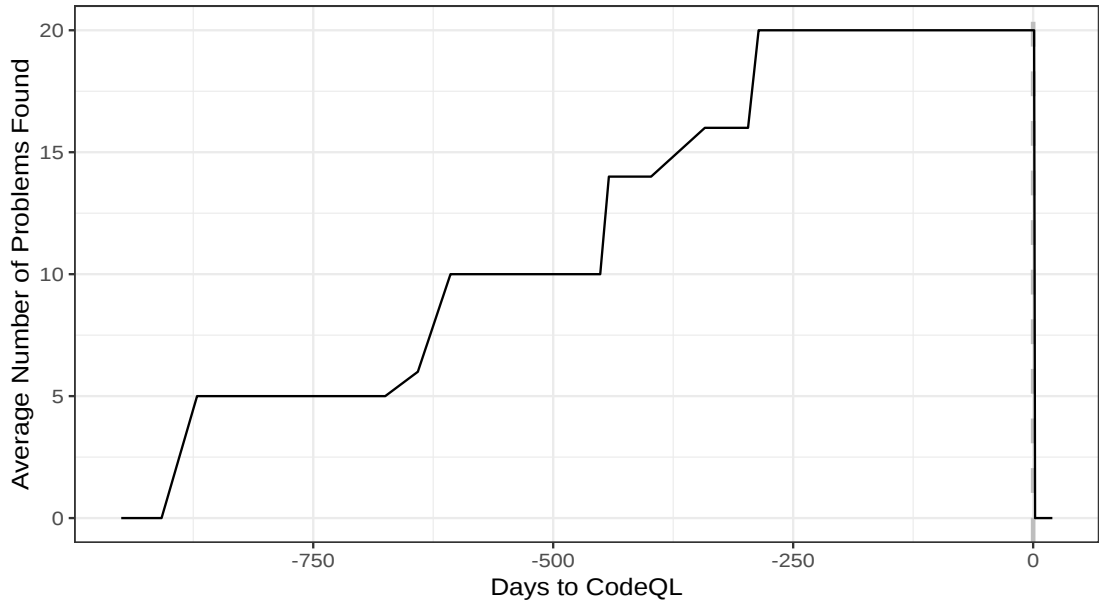

Figure 4.22: Development of “Stored XSS” issues in the repository ”livepraise” by cadimos (GitHub Repository-ID: 305507071). Period: 950 days before CodeQL activation and 20 days after.

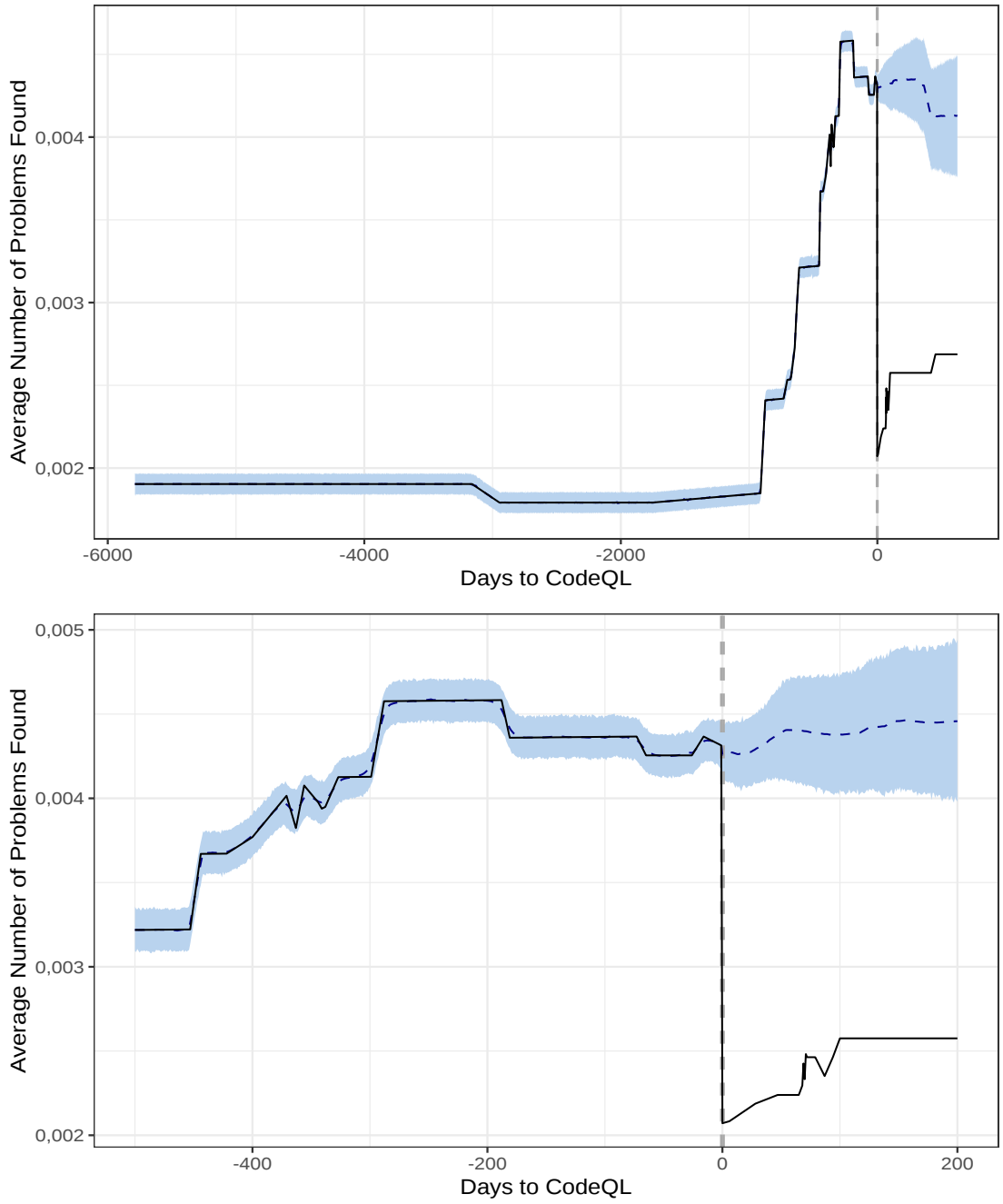

Figure 4.23: CausalImpact analysis of the rule “Stored XSS”

4.2 Impact of CodeQL Configuration on Effectiveness

As described before, CodeQL scanning can be customized via a configuration file. Thus, various parameters such as the scan frequency or the triggers for a scan may differ between repositories. Most repositories in the sample use the default configuration with weekly scans and a scan on a push or pull request. The following is a comparison of the various configurations to determine if there are certain configurations that are particularly effective.

4.2.1 Effects of the Scan Frequency

The 8,533 repositories with weekly scans had an average of 1.82 problems after CodeQL was activated, while the 398 repositories with adjusted scan intervals had slightly fewer problems at 1.56 (Figure 4.24 & 4.25). The average number of problems changed by approximately -0.040 with a 95% interval of [-0.068, -0.011] for repositories with weekly scans. This corresponds to a reduction of about 2% (95% interval: [1%, 4%]). This effect is also statistically significant (Bayesian one-sided tail-area probability p = 0.007).

Projects with a custom scan frequency show large fluctuations after CodeQL activation, owed to the small number of projects studied. In fact, the highest measured number of problems occurs after CodeQL is activated. The analysis by CausalImpact suggests a decrease of about 0.010 problems (95% interval: [-0.057, 0.038]). According to CodeQL, the probability of observing this effect by chance is 33.3% and thus this result can not be classified as statistically significant.

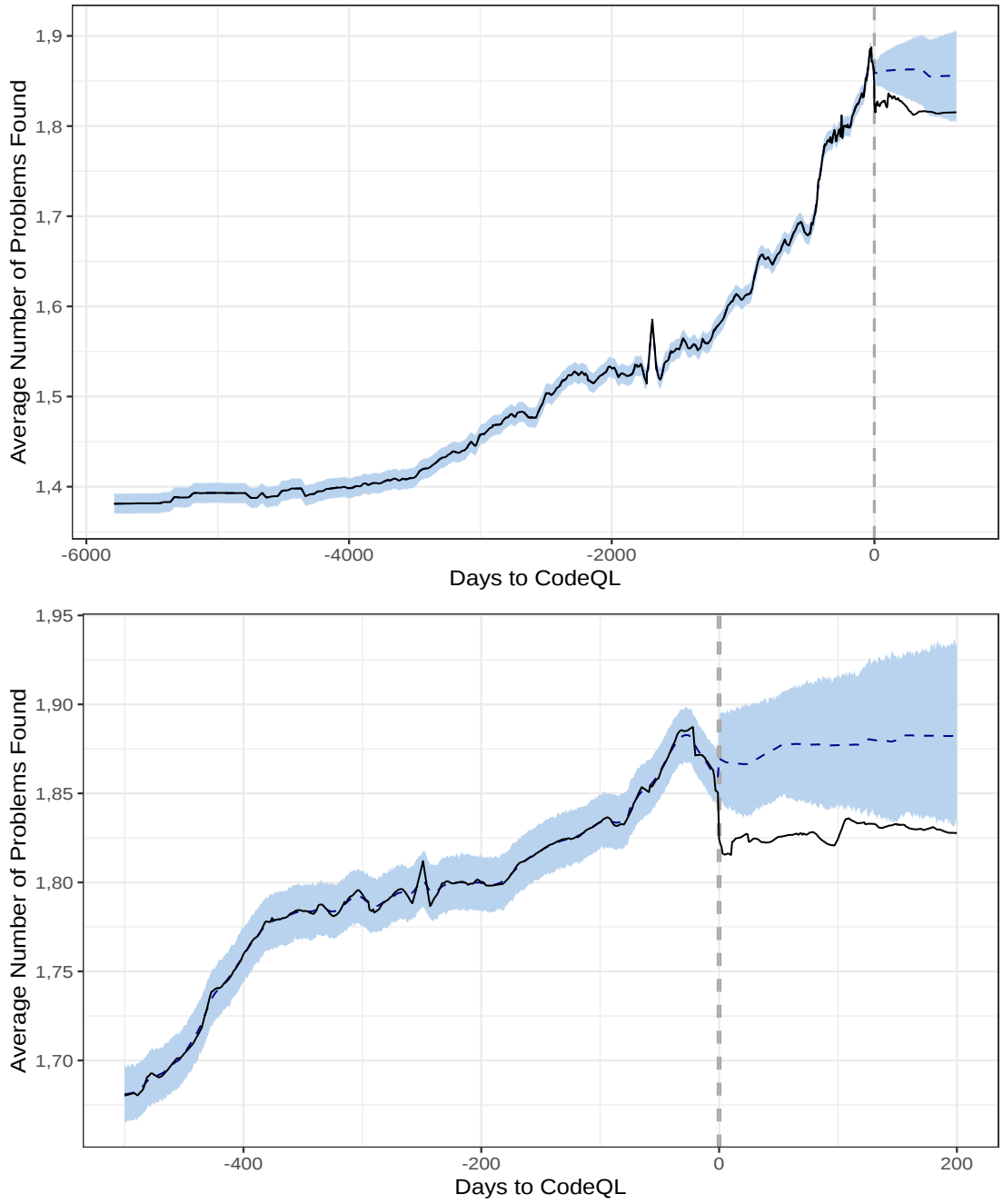

Figure 4.24: CausalImpact analysis of the overall performance across all projects with weekly CodeQL scans.

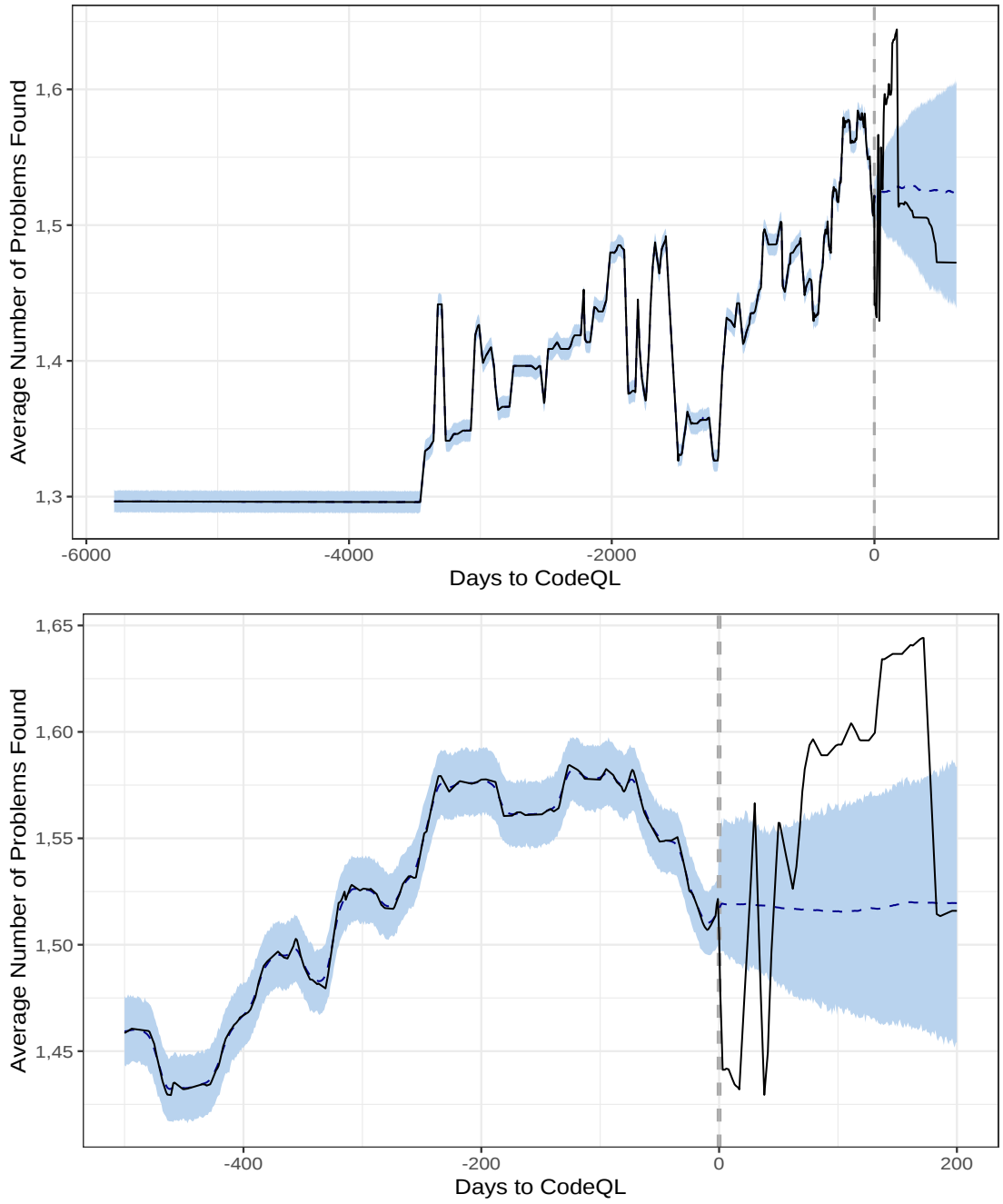

Figure 4.25: CausalImpact analysis of the overall performance across all projects that do not run weekly scans.

4.2.2 Effects of the Trigger "Push"

By default, CodeQL scans every push for specified branches. This setting is also used by 8,784 (98.3%) of the repositories. 147 repositories (1,7%) decided to disable these scans or made a change to this setting.

The 8,784 repositories scanned after each push had an average of 1.81 vulnerabilities, which is 2% (95% interval: [0%, 4%]) lower than the 1.84 defects expected by CausalImpact (Figure 4.26). Thus, maintainers resolved an average of 0.037 issues raised by CodeQL, with a 95% interval of [0.0085, 0.065]. Since the probability of observing this effect by chance is about 1%, the result is also statistically significant.

The remaining repositories had an average of 1.72 defects after CodeQL was enabled. The model calculated by CausalImpact, however, expected an average of 1.86 problems (95% interval: [1.62, 2.04]) (Figure 4.27). Thus, these projects changed the number of problems by -0.14, with a 95% interval of [-0.32, 0.032]. This corresponds to a percentage change of -8%, with a 95% interval of [-17%, +2%]. However, since the probability of observing this effect by chance is 5.1%, these results are not statistically significant. However, the sample contains only 31 repositories that showed a change in the number of problems in the period of 500 days before as well as 200 days after. More projects would need to be studied to obtain more meaningful results.

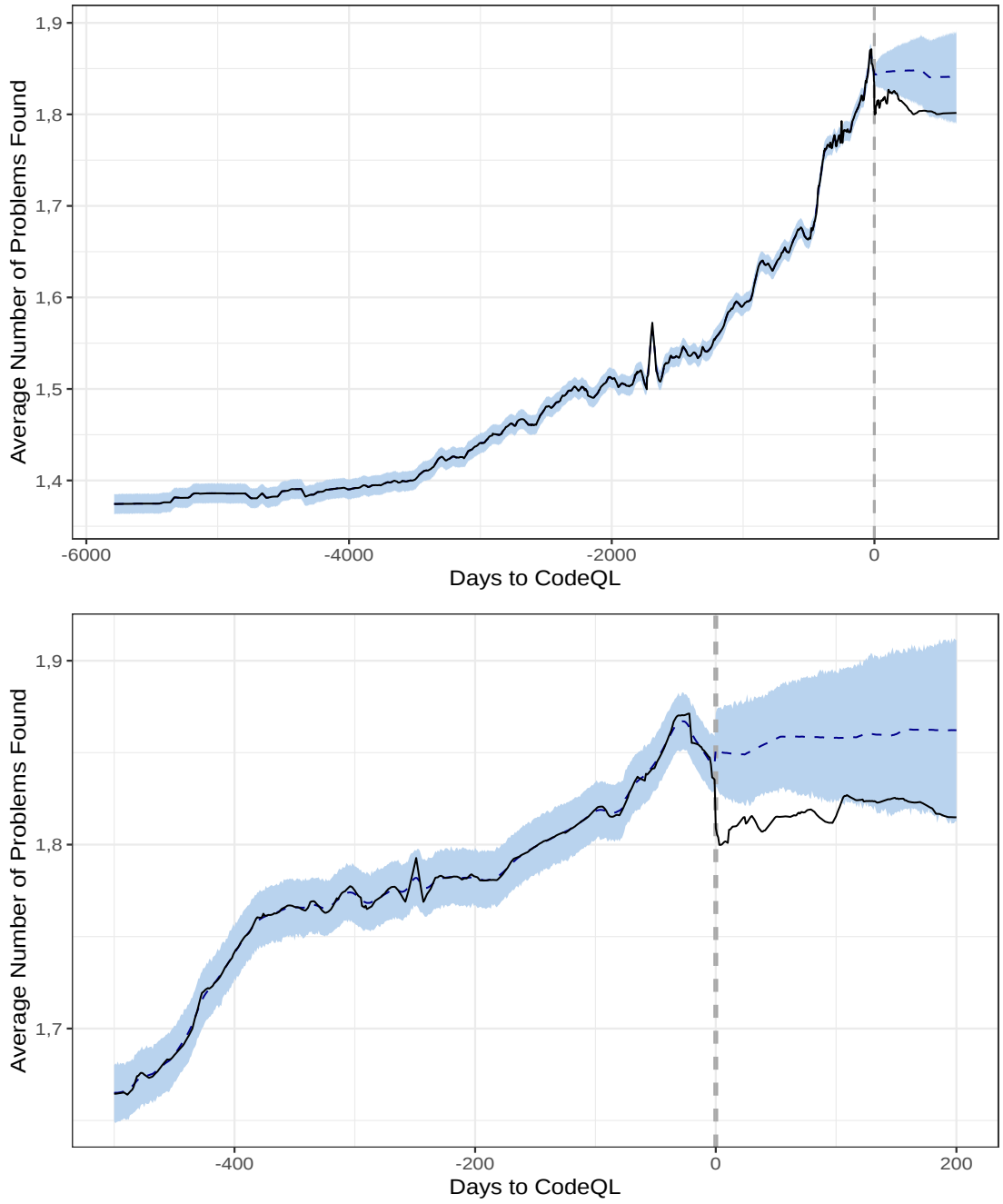

Figure 4.26: CausalImpact Analysis of the development of problems in projects scanned on push.

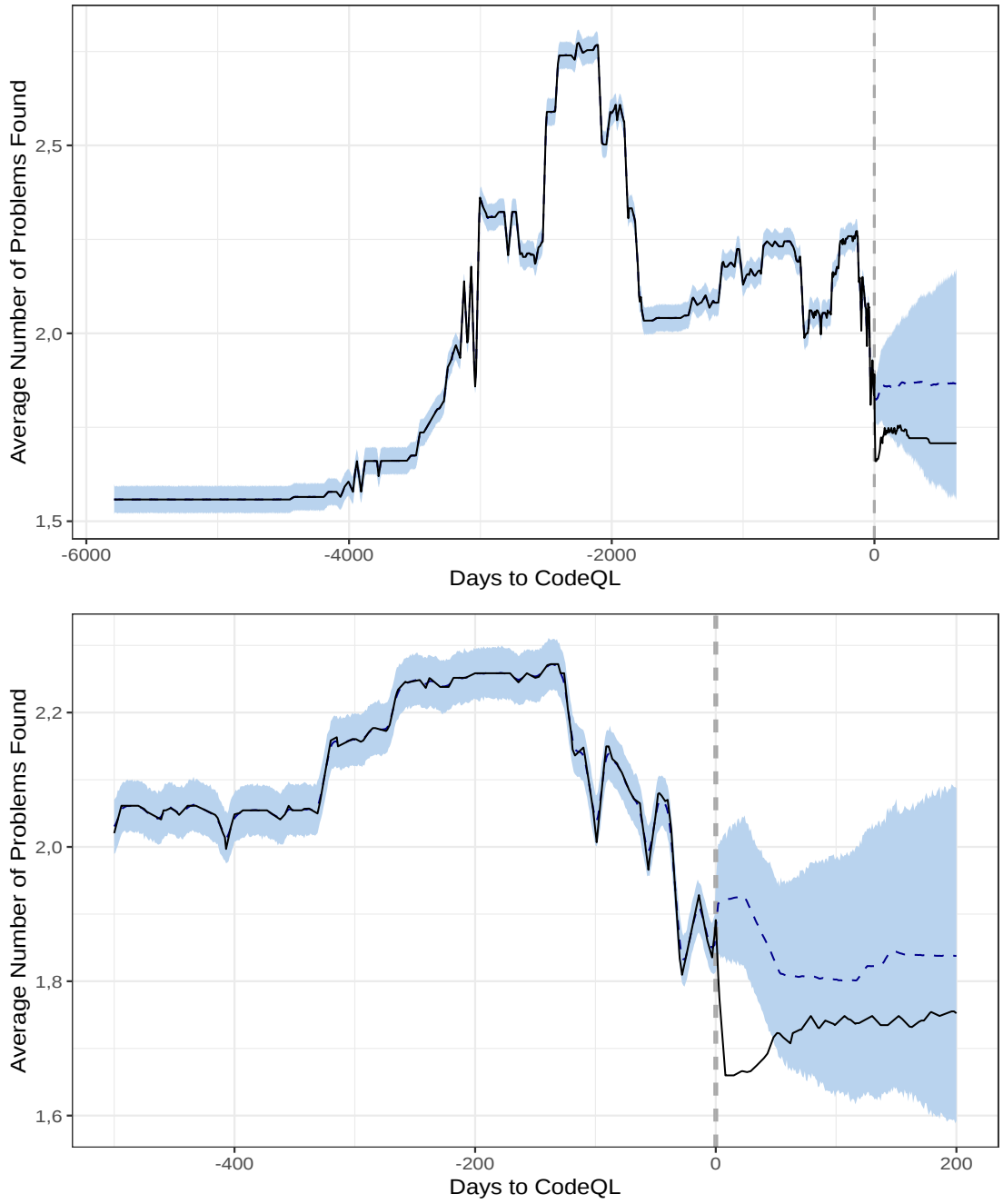

Figure 4.27: CausalImpact Analysis of the development of problems in projects that are not scanned on push or changed this setting.

4.2.3 Effects of the Trigger "Pull Request"

8,772 repositories (98.2%) use the default setting where every pull request is scanned, while 159 repositories (1.8%) deviate from this. If a new user opens a pull request at a repository with this setting enabled, a comment is added to GitHub showing the results. If the developer visits the website, he sees potential problems before merging (Figure 4.28).

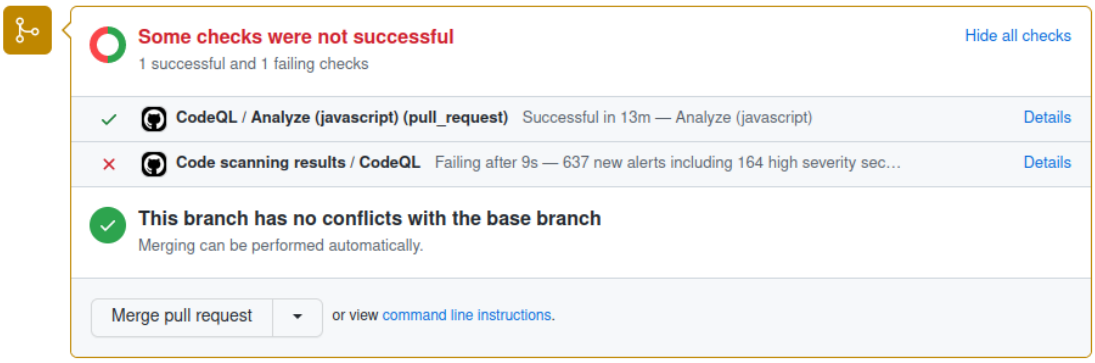

Figure 4.28: For this pull request, CodeQL warns that 637 new vulnerabilities have been identified.

The repositories where every pull request is scanned had an average of 1.83 problems in the phase after CodeQL was activated (Figure 4.29). However, if this intervention had not taken place, CausalImpact would have expected 1.86 problems, with a 95% interval of [1.83, 1.89]. This corresponds to a reduction of 2% (95% interval: [0%, 3%]) or, in absolute terms, 0.037 fixed problems. Since the probability of observing this effect by chance is 0.01, the improvement is statistically significant.

The remaining repositories had fewer problems on average, approximately 0.71 defects were found after CodeQL was enabled (Figure 4.30). Since an average of 0.76 problems was actually expected for this period, this indicates a change of -7%, the 95% interval being [-31%, +17%]. This corresponds to an average of 0.055 solved problems per repository (95% interval: [-0.23, +0.13]). The probability of obtaining this effect by chance is 27.1%, which means that it is not a statistically significant finding.

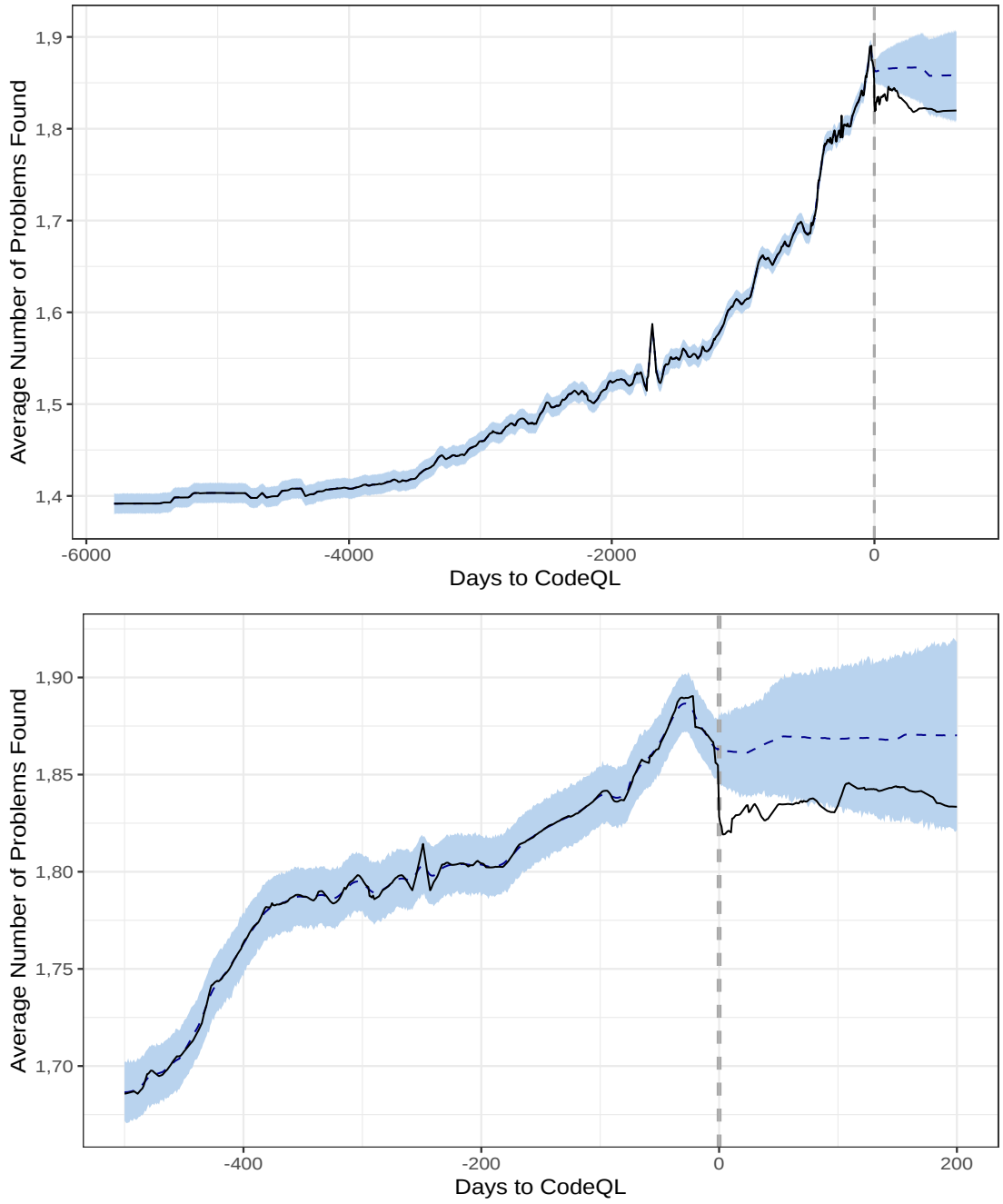

Figure 4.29: CausalImpact analysis of the development of problems in projects scanned on a pull request.

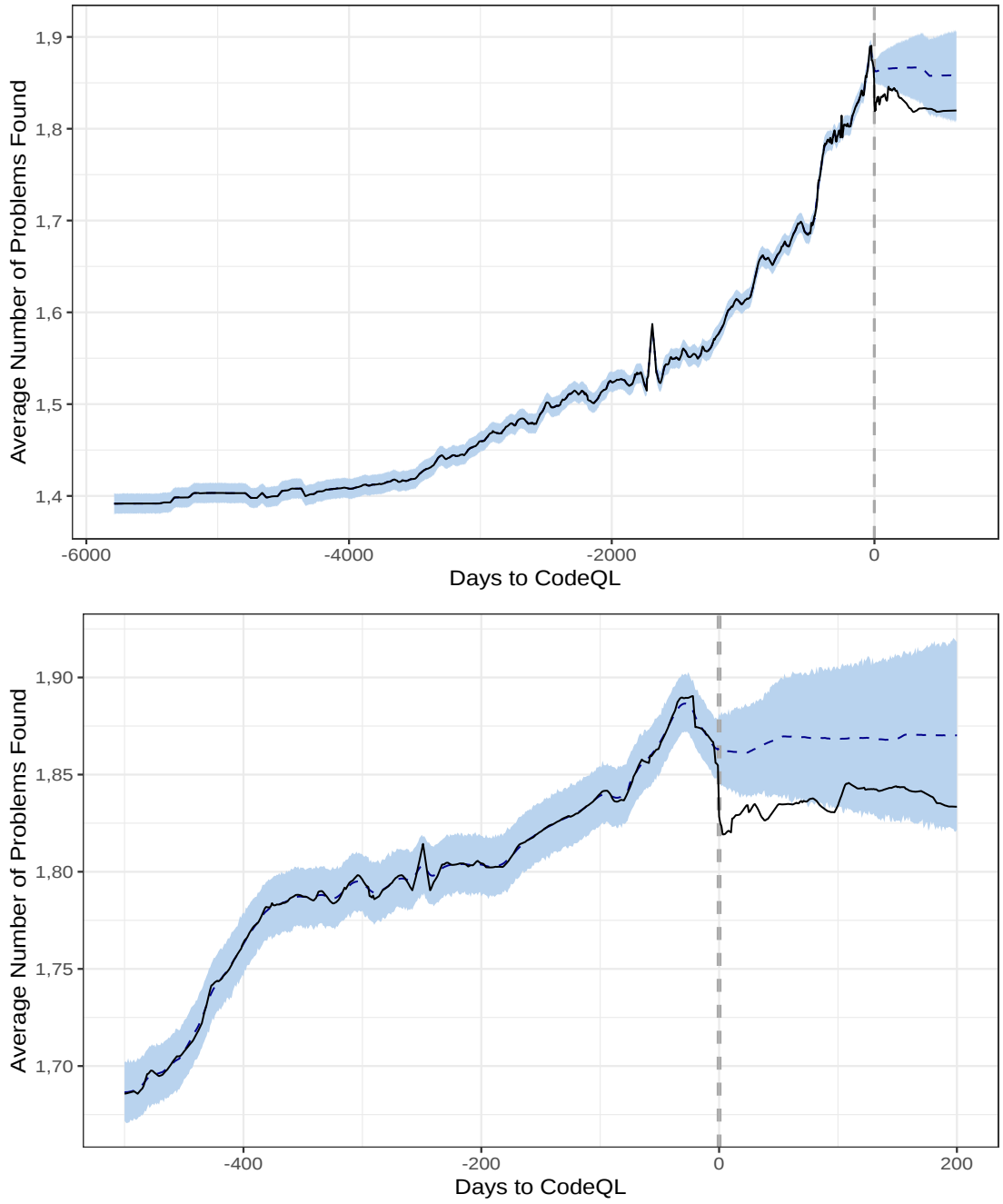

Figure 4.30: CausalImpact analysis of the development of problems in projects that deactivated pull request scanning.

4.3 Breakdown by Repository Characteristics

4.3.1 Number of Repository Stargazers

GitHub users can star repositories to track projects they find interesting. To a certain extent, the number of stars can be used to draw conclusions about the popularity of a repository. The sample includes 7,432 projects with 10 or fewer stars and 1,509 repositories with more than 10 stars on GitHub.

Projects with more than 10 stars had an average of 2.45 defects after CodeQL was enabled, which is slightly lower than the expected 2.46 problems (95% interval: [2.39, 2.52]) (Figure 4.31). This corresponds to a change of -0.0065 problems, or 0%. The 95% intervals are [-0.067, 0.055] and [-3%, +2%]. The probability of a causal effect is only 59%, making the results not statistically significant.

In contrast, the results of repositories with 10 or fewer stars are more significant: After CodeQL was enabled, an average of 1.68 vulnerabilities were found. Without this intervention, however, CausalImpact would have expected an average of 1.72 problems (Figure 4.32). Thus, approximately 0.043 issues were fixed per repository, with a 95% interval of [0.017, 0.069]. This corresponds to an improvement of 3% (95% interval: [1%, 4%]). This effect is causal with a probability of 99.9% and thus statistically significant.

The projects with 10 or fewer stars had an average of about 204kB and the other projects had about 431kB of JavaScript code.

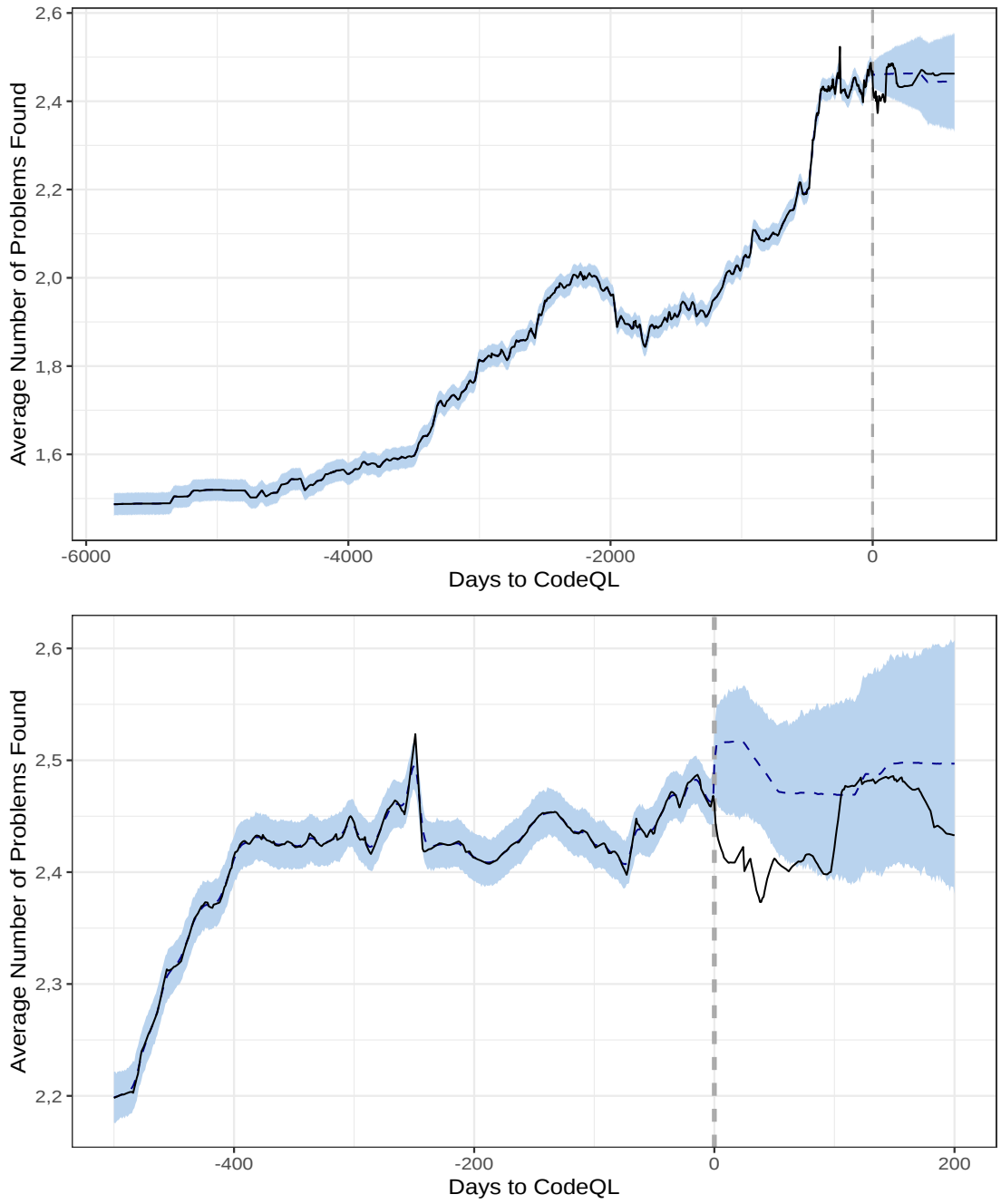

Figure 4.31: CausalImpact analysis of the development of defects in projects with more than 10 stars.

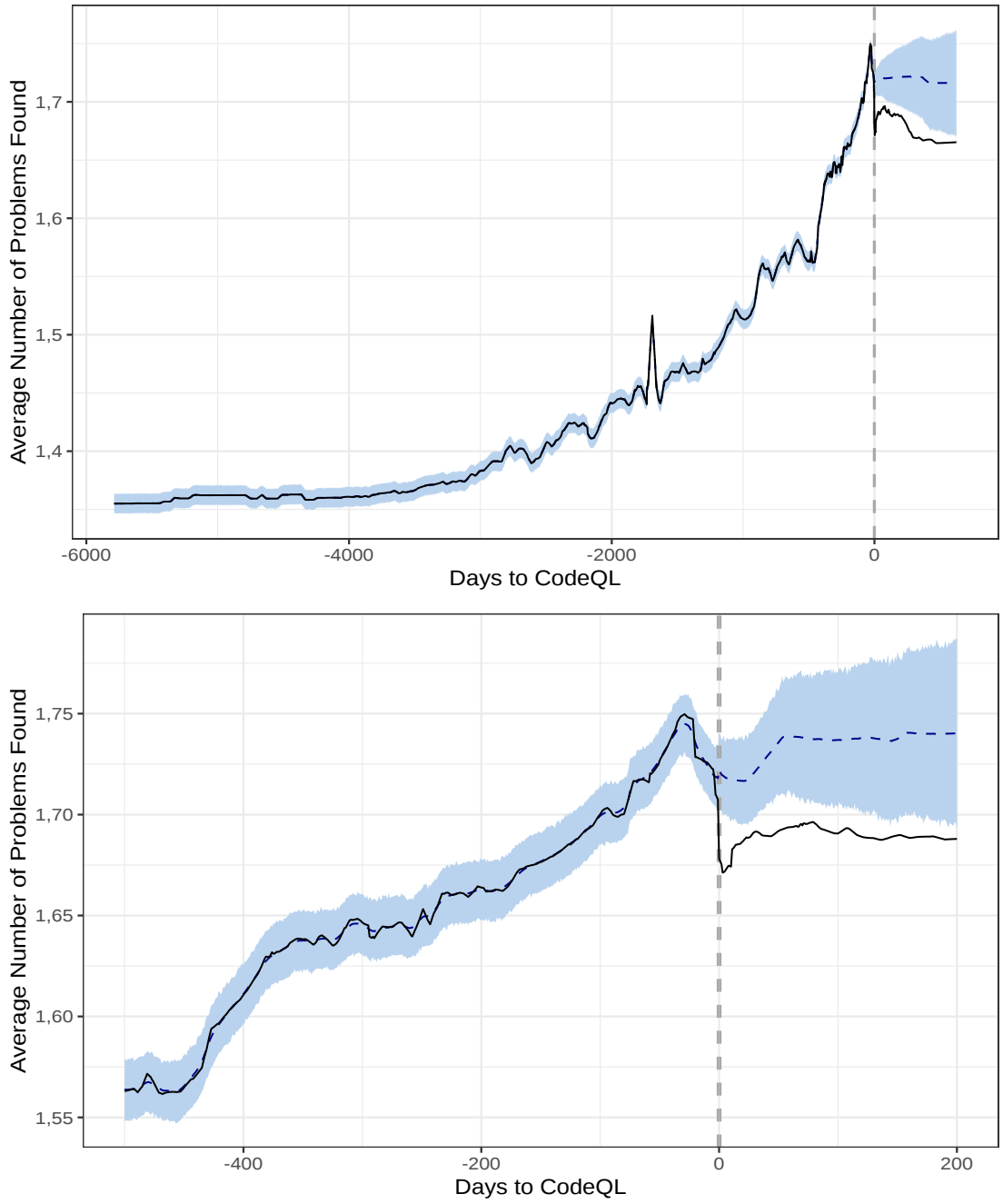

Figure 4.32: CausalImpact analysis of vulnerability development in projects with 10 or less stars on GitHub.

4.3.2 Number of Contributors to a Repository

On average, 9.7 users contributed to a repository, the median being 2. 2,270 repositories had more than 4 contributors, for these repositories no statistically significant change due to CodeQL could be detected (Figure 4.33). Thus, an average of 2.59 problems were found after CodeQL was enabled. The counterfactual prediction had calculated 2.60 problems for this time period. As a result, an average of -0.017 problems were fixed, which corresponds to a change of -1%. The 95% intervals are [-0.083, 0.050] and [-3%, +2%]. However, with a probability of 31.6%, this effect was obtained only by chance.

Projects with 4 or fewer contributors had an average of 1.54 issues after CodeQL was enabled, whereas 1.59 would have been expected without the intervention (Figure 4.34). Thus, an average of 0.047 issues were resolved (95% interval: [0.022, 0.072]). This corresponds to an improvement of 3%, with a 95% interval of [1%, 5%]. The probability of observing this effect only by chance is p=0.001, making the result statistically significant.

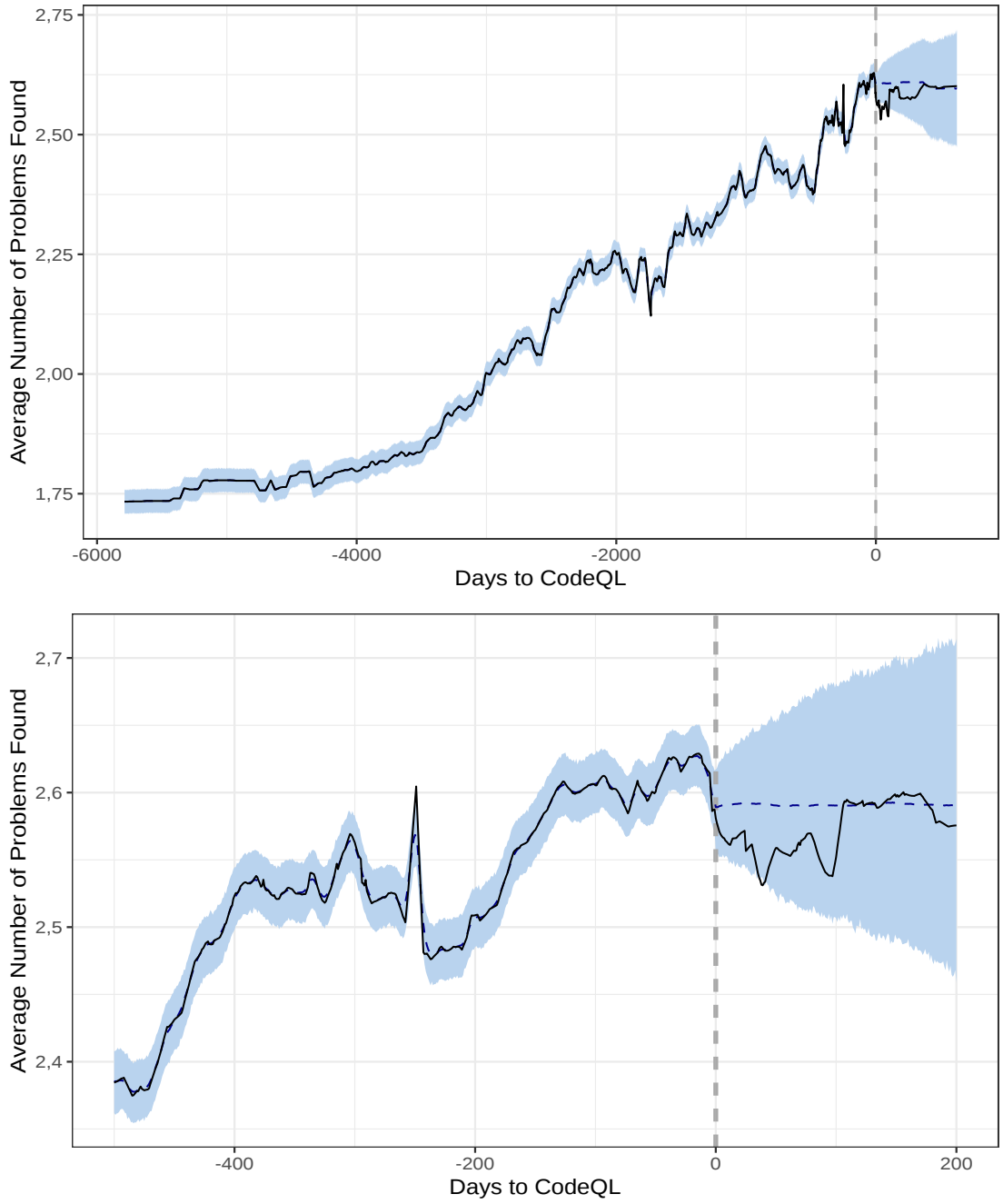

Figure 4.33: CausalImpact analysis of the development of defects in projects with more than 4 contributors.

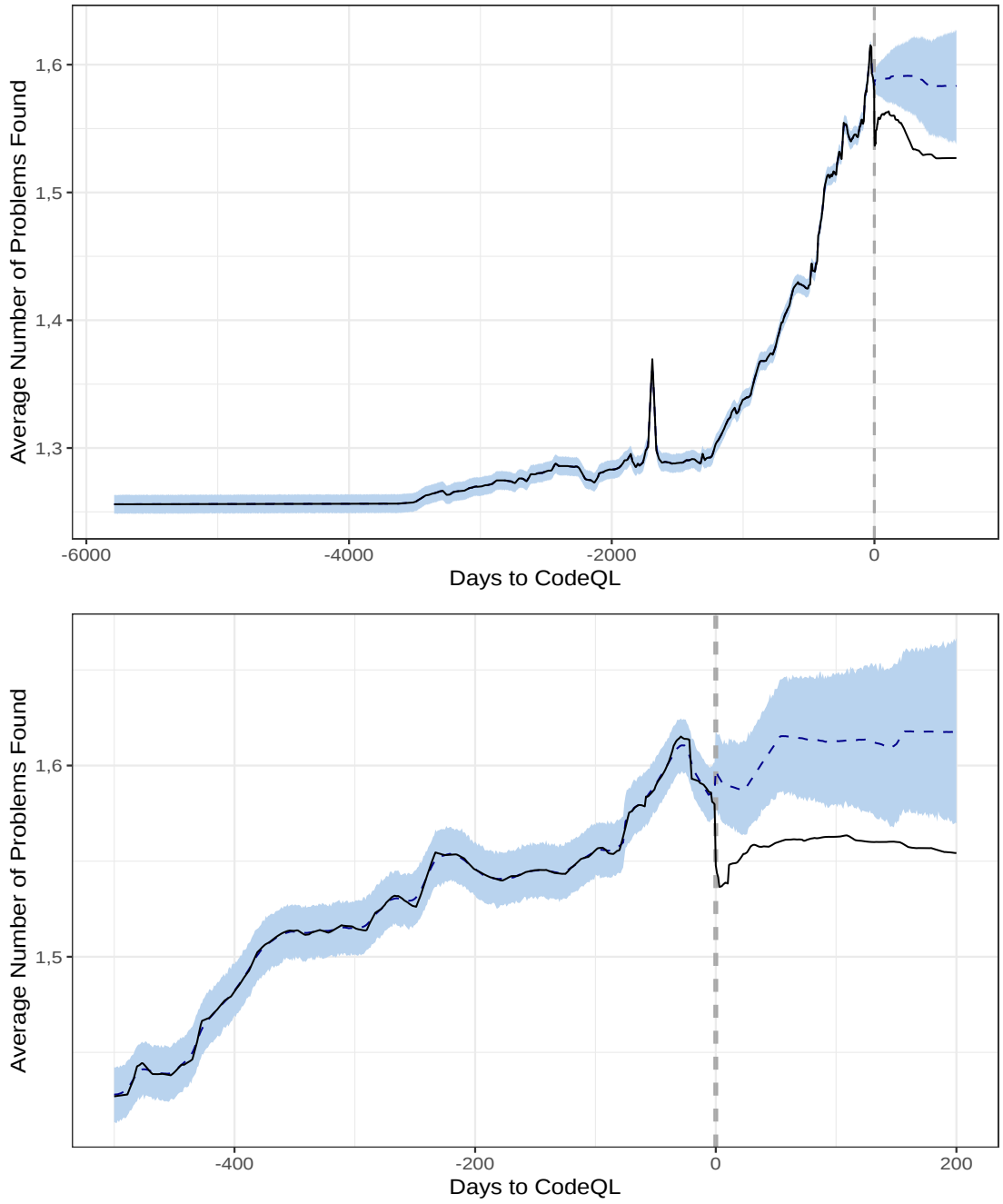

Figure 4.34: CausalImpact analysis of the development of defects in projects with 4 or fewer contributors.

4.3.3 Projects by Performance

410 repositories improved and 538 repositories worsened in the 500-day period before and 200 days after CodeQL was activated. 7,983 projects that did not change at all were not taken into account in the following analysis.

For the top projects, an average of 4.43 issues were identified after CodeQL was activated (Figure 4.35). However, without the intervention, 7.09 issues would have been expected, with a 95% interval for this prediction of [6.73, 7.44]. Thus, maintainers fixed an average of 2.23 vulnerabilities (95% interval: [1.75, 2.76]). This corresponds to a reduction of 33%, where the 95% interval is [26%, 40%].

The trend is the other way round for the 538 repositories that perform worse in the first 200 days with CodeQL compared to the 500 days before (Figure 4.36). Thus, the average number of issues after the intervention is approximately 11.57, although 10.19 were expected for this period (95% interval: 9.87, 10.52). This corresponds to 1.37 additional defects, which means that CodeQL has increased the number of problems by about 13%. The 95% intervals for these results are [1.05, 1.69] and [10%, 17%]. The probability of observing this effect by chance is only 1%, which makes it statistically significant.

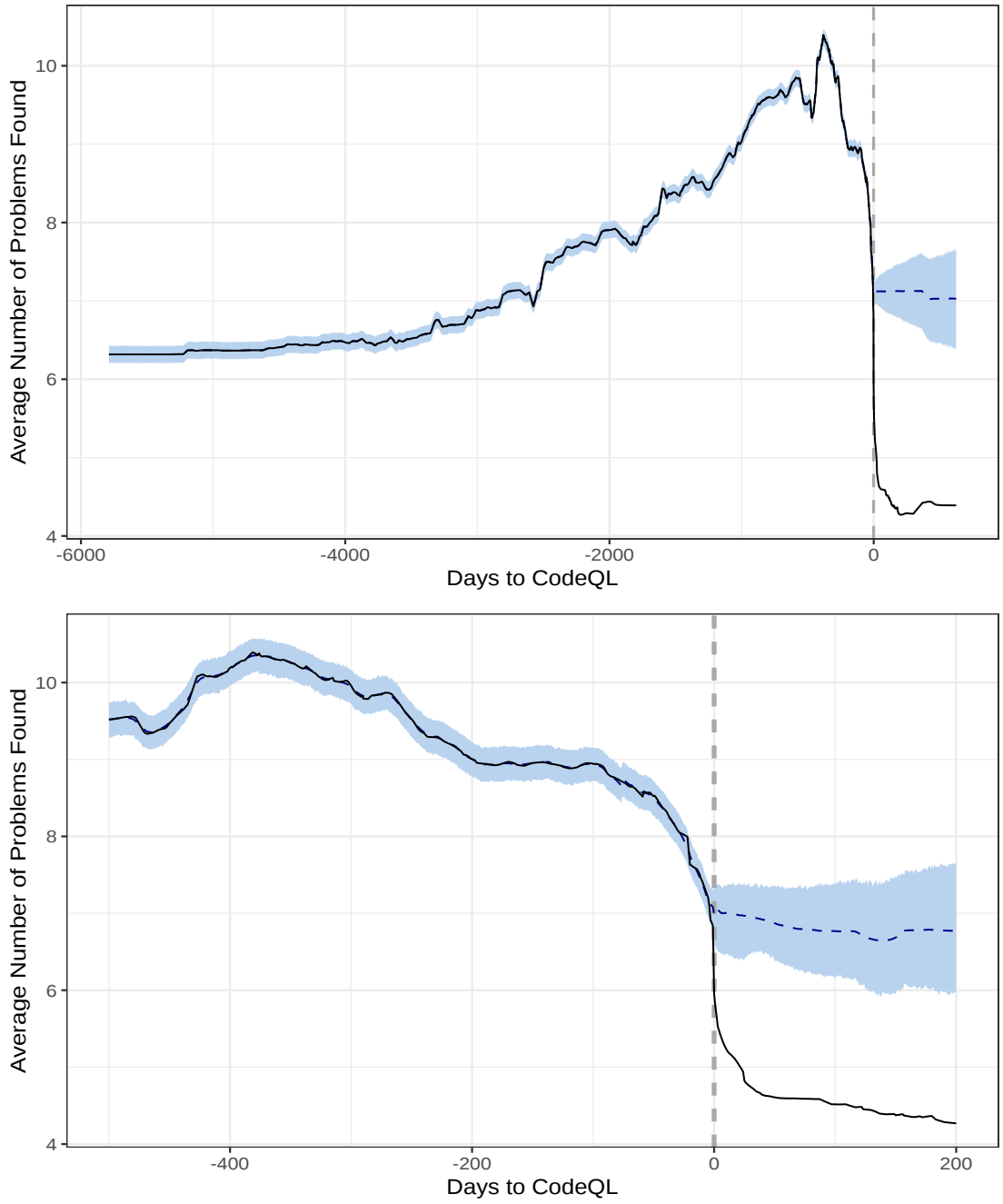

Figure 4.35: CausalImpact analysis of the development of defects in projects that improved in the 200 days after CodeQL was activated compared to 500 days before. Please note that these graphs do not include those projects where no problems were found. Accordingly, the average is also significantly higher.

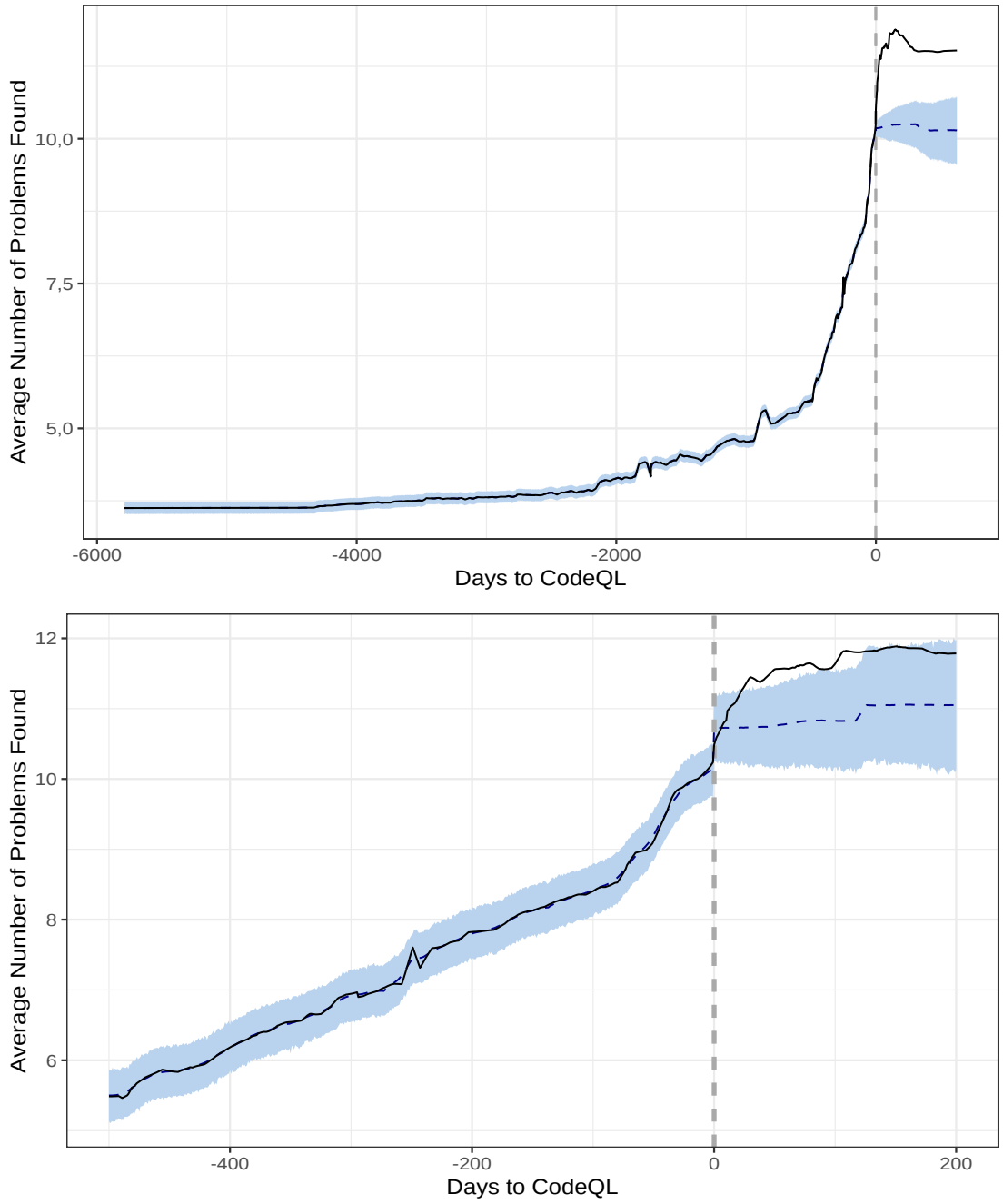

Figure 4.36: CausalImpact analysis of the development of vulnerabilities in projects that worsened in the 200 days after CodeQL was activated compared to 500 days before. Please note that these graphs do not include those projects where no problems were found. Accordingly, the average is also significantly higher.

5 Discussion

The results show that CodeQL reduces the average number of problems. However, with a reduction of only 2%, the effect is not sufficiently large to produce a satisfactory change on its own. It is clear that much greater efforts are still needed to address the alarmingly large spread of security vulnerabilities.

This study examined only repositories that have CodeQL enabled. However, it can be assumed that maintainers who enable CodeQL also place an increased emphasis on code security. Nevertheless, the effects of CodeQL differ greatly depending on the repository:

	Repositories
No Problems Found	7,337 (82.2%)
Problems Found, No Change	1,077 (12.1%)
Problems Found, Worsened	262 (2.9%)
Problems Found, Improved	254 (2.8%)
Total	8,931

Table 5.1: Data refers to development 45 days before vs after activation of CodeQL.

This confirms the thesis that has already been put forward in other papers: **It is easier to identify a security vulnerability than to fix it** (Alomar et al. 2020; OpenSSF 2022).

It is also noticeable that there are some outliers where an increase in vulnerabilities coincides with the activation of CodeQL (Figure 4.36 & 4.15). The reasons for this can be manifold. For example, the development team may have been particularly active during this period and made many changes to the code base, which also made it more likely that new vulnerabilities would be added. However, a plausible explanation could be that the activation of CodeQL gave a false sense of security. Especially if the development team did not notice the newly found security vulnerabilities (see last point at "Possible Causes Limiting the Efficiency of CodeQL").

It is noticeable that CodeQL is more effective for less popular repositories and those with only a few contributors (See "Number of Contributors to a Repository" & "Number of Repository Stargazers"). A possible reason could be that no one feels

responsible for fixing the security vulnerabilities in the other projects.

5.1 Long Term Effects

Immediately after CodeQL is enabled, the biggest drop in security vulnerabilities occurs. Later on, the trend tends to be downward, depending on the rule, but nowhere near as significant as immediately after activation. One possible reason for this is that the development team tackles easy-to-solve problems head-on and skips the more complex ones. For a discussion on why developers may not fix a vulnerability, see below.

5.2 Effects of CodeQL Configuration on Efficiency

94.5% of the examined projects use the default scan settings suggested by GitHub, with weekly scans and on push and pull requests. This indicates that the workflow settings suggested by GitHub are either satisfactory for most users, or that the necessary knowledge for customization is lacking. The analyses suggest that the efficiency of CodeQL is not, or at least only weakly, dependent on the specific scan settings. This indicates that the configuration of CodeQL has less to do with the efficiency of the tool than with the individual preferences of the developer.

5.3 Possible Causes Limiting the Efficiency of CodeQL

The results strongly suggest that the effectiveness of GitHub's security interventions is centrally dependent on the team of maintainers. Investigating causes for the large discrepancy in repositories is not a central part of this paper. However, the following possible causes provide an outlook for future research:

- **Maintainers decide that fixing a security vulnerability is not worth the effort ("won't fix") or postpone the problem to a later point in time.**
  The potential damage of a security vulnerability may be too small to justify an elaborate bug fixing process. This is particularly the case if a tool is only intended for certain scenarios, for example as an internal application.

  In addition, unsafe code can also be part of libraries or components that the repository uses. Developers may thus be hesitant to change the code base and risk breaking it. Inexperienced developers, in particular,

may also underestimate the danger that insecure code can pose to both operators and end users.

Especially with a large number of problems found, triage is necessary. Since the queries examined in this study are only classified as warnings, other critical defects may be fixed with higher priority.

- **The problem is not applicable or it is a false alarm.** Although the documentation of CodeQL states the precision of the examined queries as high, false positives were most likely also identified. For instance, some vulnerabilities may even be intentional, for example as part of test cases, where they do not pose a real threat.
  Since GitHub does not provide comprehensive data on how many of the code scanning alerts are dismissed, this effect is difficult to quantify.

- **Lack of skill or knowledge on how to fix the issue.**
  Some developers may not understand the exact program structure, especially if they are new contributors or they copied code fragments from other sources, such as Stack Overflow or another open source project. In addition, the description and recommendations provided by GitHub may be incomprehensible to some users.
  There are many other potential scenarios where a maintainer would like to fix a problem, but needs additional support and assistance to do so. A recent study by Bandara et al. (2021) also found that 16% of the analyzed commits that were intended to close a vulnerability also added at least one new issue.

- **The developers are not aware of the problems found by CodeQL.**
  For security reasons, only users with write permissions to the repository see the alerts. Some of those users may not be aware that their repository contains security vulnerabilities. Possible reasons for this could be an unintuitive site design (Figure 5.1 & 5.2) coupled with inexperienced developers. Although it is a native GitHub tool, the code scanning alerts are probably not seamlessly integrated into the development workflow of most maintainers. However, this suspicion has yet to be verified and further research is needed.
  For pull requests, the results of CodeQL are displayed as comments (Figure 4.28). The likelihood that these problems will be overlooked is therefore considered to be lower.

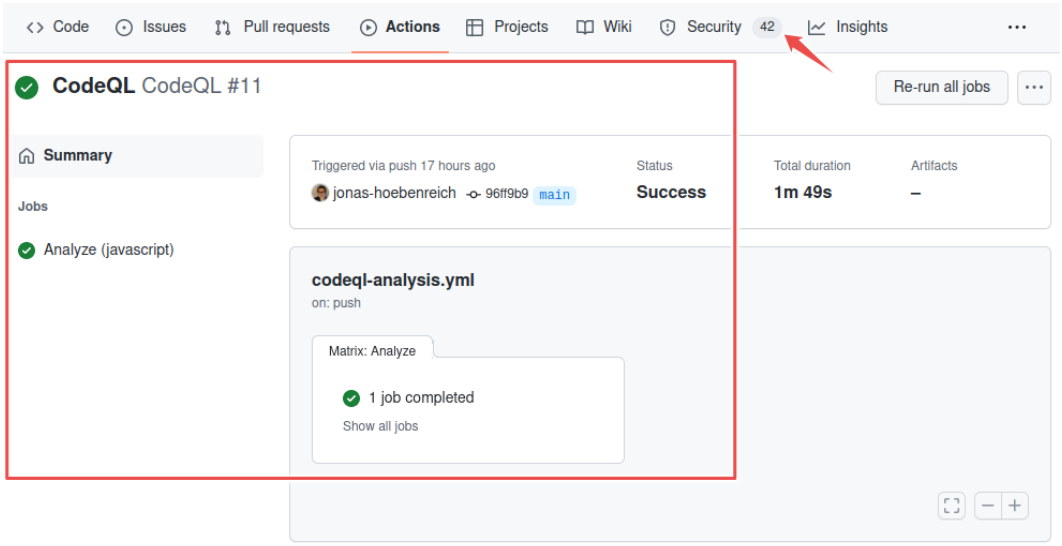

Figure 5.1: The current design may give some inexperienced users the wrong impression that the CodeQL analysis ran successfully, but no problems were found. The red rectangle marks the information that shows that the workflow ran without errors, possibly deceiving the user into a false sense of security. The fact that 42 problems were found during the analysis is just a rather inconspicuous note next to "Security" (marked by red arrow).

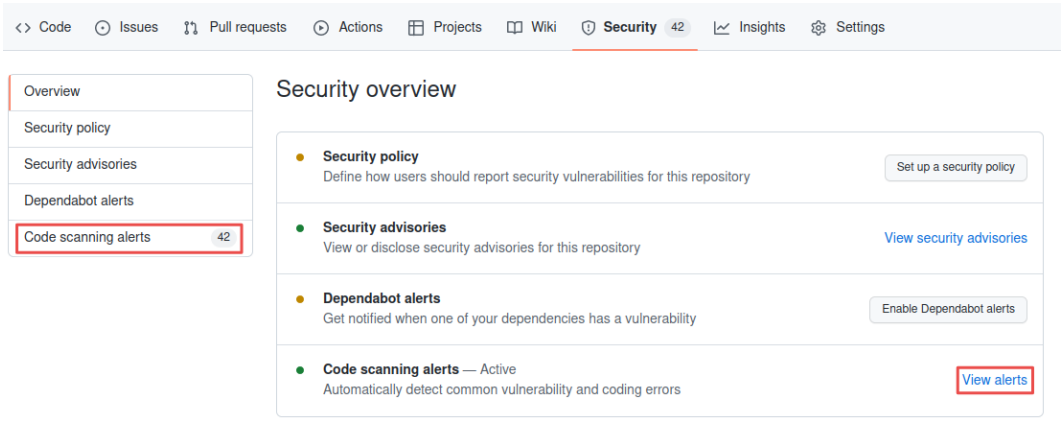

Figure 5.2: Even in the Security tab, the findings are not displayed upfront. The user has to click on "Code scanning alerts" or "view alerts" (marked by red boxes) before he can view the issues.

5.4 Prospects

In February 2022, GitHub announced an improved version of CodeQL that leverages machine learning to identify additional results[1]. This experimental feature will also be applied to cross-site scripting type issues. This study does not currently take this new development into account. Future research could investigate whether this feature increases the efficiency of CodeQL.

However, this study identified a problem that even this update does not address: **Most of the security vulnerabilities identified by CodeQL are not patched.** Therefore, additional strategies are needed to help developers deal with security vulnerabilities more responsibly. The Open Source Software Security Mobilization Plan calls for offering security advisory and support by experts to maintainers of the top 10,000 open source repositories. The plan also includes the establishment of an open source security incident response team to assist projects with time-critical security incidents (OpenSSF 2022).

In summary, code analyzers are only one aspect of software security. In the end, real people are still needed to analyze and ultimately fix a problem.

[1] https://github.blog/2022-02-17-code-scanning-finds-vulnerabilities-using-machine-learning/

List of Figures

List of Tables

Bibliography

Alomar, N., P. Wijesekera, E. Qiu, and S. Egelman (Aug. 2020). ""You've Got Your Nice List of Bugs, Now What?" Vulnerability Discovery and Management Processes in the Wild." In: *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*. USENIX Association, pp. 319–339. ISBN: 978-1-939133-16-8. URL: https://www.usenix.org/conference/soups2020/presentation/alomar.

Avgustinov, P., A. I. Baars, A. S. Henriksen, G. Lavender, G. Menzel, O. De Moor, M. Schafer, and J. Tibble (May 2015). "Tracking static analysis violations over time to capture developer characteristics." In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence: IEEE.

Bandara, V., T. Rathnayake, N. Weerasekara, C. Elvitigala, K. Thilakarathna, P. Wijesekera, K. De Zoysa, and C. Keppitiyagama (Nov. 2021). "Demo: Large scale analysis on vulnerability remediation in open-source JavaScript projects." In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event Republic of Korea: ACM.

Bandara, V., T. Rathnayake, N. Weerasekara, C. Elvitigala, K. Thilakarathna, P. Wijesekera, and C. Keppitiyagama (Sept. 2020). "Fix that Fix Commit: A real-world remediation analysis of JavaScript projects." In: *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Adelaide, Australia: IEEE.

CISA (Jan. 2022). *Implement Cybersecurity Measures Now to Protect Against Potential Critical Threats*. Cybersecurity & Infrastructure Security Agency. URL: https://www.cisa.gov/sites/default/files/publications/CISA_Insights-Implement_Cybersecurity_Measures_Now_to_Protect_Against_Critical_Threats_508C.pdf.

CSIS (2022). *Significant Cyber Incidents*. Center for Strategic and International Studies. URL: https://www.csis.org/programs/strategic-technologies-program/significant-cyber-incidents (visited on 06/15/2022).

Johnson, B., Y. Song, E. Murphy-Hill, and R. Bowdidge (May 2013). "Why don't software developers use static analysis tools to find bugs?" In: *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE.

NCSC (Feb. 2018). *Russian military 'almost certainly' responsible for destructive 2017 cyber attack*. National Cyber Security Centre. URL: https://www.ncsc.gov.uk/news/

russian-military-almost-certainly-responsible-destructive-2017-cyber-attack (visited on 06/15/2022).

OpenSSF (May 2022). *The Open Source Software Security Mobilization Plan*. Open Source Security Foundation. URL: https://openssf.org/oss-security-mobilization-plan/.

Source code and data of this thesis are available at https://github.com/jonas-hoebenreich/effectiveness-of-codeql.